

Electron Documentation

目錄

Electron 中文文档

> 来源：Electron 中文文档

常见问题

Electron 常见问题

Electron 会在什么时候升级到最新版本的 Chrome？

通常来说，在稳定版的 Chrome 发布后两周内，我们会更新 Electron 内的 Chrome 版本。

我们只会使用 stable 版本的 Chrome。但如果在 beta 或 dev 版本中有一个重要的更新，我们会把补丁应用到现版本的 Chrome 上。

Electron 会在什么时候升级到最新版本的 Node.js？

我们通常会在最新版的 Node.js 发布后一个月左右将 Electron 更新到这个版本的 Node.js。我们通过这种方式来避免新版本的 Node.js 带来的 bug（这种 bug 太常见了）。

Node.js 的新特性通常是由新版本的 V8 带来的。由于 Electron 使用的是 Chrome 浏览器中附带的 V8 引擎，所以 Electron 内往往已经 有了部分新版本 Node.js 才有的崭新特性。

如何在两个网页间共享数据？

在两个网页（渲染进程）间共享数据最简单的方法是使用浏览器中已经实现的 HTML5 API，比较好的方案是用 Storage API， `localStorage` ， `sessionStorage` 或者 IndexedDB。

你还可以用 Electron 内的 IPC 机制实现。将数据存在主进程的某个全局变量中，然后在多个渲染进程中使用 `remote` 模块来访问它。

```
// 在主进程中
global.sharedObject = {
  someProperty: 'default value'
};
```

```
// 在第一个页面中
require('remote').getGlobal('sharedObject').someProperty = 'new value';
```

```
// 在第二个页面中
console.log(require('remote').getGlobal('sharedObject').someProperty);
```

为什么应用的窗口、托盘在一段时间后不见了？

这通常是因为用来存放窗口、托盘的变量被垃圾收集了。

你可以参考以下两篇文章来了解为什么会遇到这个问题。

- 内存管理
- 变量作用域

如果你只是要一个快速的修复方案，你可以用下面的方式改变变量的作用域，防止这个变量被垃圾收集。

从

```
app.on('ready', function() {
  var tray = new Tray('/path/to/icon.png');
})
```

改为

```
var tray = null;
app.on('ready', function() {
  tray = new Tray('/path/to/icon.png');
})
```

在 Electron 中，我为什么不能用 jQuery、RequireJS、Meteor、AngularJS？

因为 Electron 在运行环境中引入了 Node.js，所以在 DOM 中有一些额外的变量，比如 `module`、`exports` 和 `require`。这导致 了许多库不能正常运行，因为它们也需要将同名的变量加入运行环境中。

我们可以通过禁用 Node.js 来解决这个问题，用如下的方式：

```
// 在主进程中
var mainWindow = new BrowserWindow({
  webPreferences: {
    nodeIntegration: false
  }
});
```

假如你依然需要使用 Node.js 和 Electron 提供的 API，你需要在引入那些库之前将这些变量重命名，比如：

```
<head>
<script>
// 重命名 Electron 提供的 require
window.nodeRequire = require;
delete window.require;
delete window.exports;
delete window.module;
</script>
<script type="text/javascript" src="jquery.js"></script>
</head>
```

为什么 `require('electron').xxx` 的结果是 undefined？

在使用 Electron 的提供的模块时，你可能会遇到和以下类似的错误：

```
> require('electron').webFrame.setZoomFactor(1.0);
Uncaught TypeError: Cannot read property 'setZoomLevel' of undefined
```

这是因为你在项目中或者在全局中安装了npm 上获取的 `electron` 模块，它把 Electron 的内置模块覆写了。

你可以通过以下方式输出 `electron` 模块的路径来确认你是否使用了正确的模块。

```
console.log(require.resolve('electron'));
```

确认以下它是不是像下面这样的：

```
"/path/to/Electron.app/Contents/Resources/atom.asar/renderer/api/lib/exports/electron.js"
```

假如输出的路径类似于 `node_modules/electron/index.js` ，那么你需要移除或者重命名 npm 上的 `electron` 模块。

```
npm uninstall electron
npm uninstall -g electron
```

如果你依然遇到了这个问题，你可能需要检查一下拼写或者是否在错误的进程中调用了这个模块。比如， `require('electron').app` 只能在主进程中使用, 然而 `require('electron').webFrame` 只能在渲染进程中使用。

向导

支持的平台

以下的平台是 Electron 目前支持的：

OS X

对于 OS X 系统仅有64位的二进制文档，支持的最低版本是 OS X 10.8。

Windows

仅支持 Windows 7 及其以后的版本，之前的版本中是不能工作的。

对于 Windows 提供 `x86` 和 `amd64` (x64) 版本的二进制文件。需要注意的是 `ARM` 版本的 Windows 目前尚不支持.

Linux

预编译的 `ia32` (`i686`) 和 `x64` (`amd64`) 版本 Electron 二进制文件都是在 Ubuntu 12.04 下编译的，`arm` 版的二进制文件是在 ARM v7（硬浮点 ABI 与 Debian Wheezy 版本的 NEON）下完成的。

预编译二进制文件是否能够运行，取决于其中是否包括了编译平台链接的库，所以只有 Ubuntu 12.04 可以保证正常工作，但是以下的平台也被证实可以运行 Electron 的预编译版本：

- Ubuntu 12.04 及更新
- Fedora 21
- Debian 8

应用部署

为了使用 Electron 部署你的应用程序，你存放应用程序的文件夹需要叫做 `app` 并且需要放在 Electron 的 资源文件夹下（在 OS X 中是指 `Electron.app/Contents/Resources/` ，在 Linux 和 Windows 中是指 `resources/` ） 就像这样：

在 OS X 中:

```
electron/Electron.app/Contents/Resources/app/
├── package.json
├── main.js
└── index.html
```

在 Windows 和 Linux 中:

```
electron/resources/app
├── package.json
├── main.js
└── index.html
```

然后运行 `Electron.app` （或者 Linux 中的 `electron` ，Windows 中的 `electron.exe` ），接着 Electron 就会以你的应用程序的方式启动。 `electron` 文件夹将被部署并可以分发给最终的使用者。

将你的应用程序打包成一个文件

除了通过拷贝所有的资源文件来分发你的应用程序之外，你可以可以通过打包你的应用程序为一个 asar 库文件以避免暴露你的源代码。

为了使用一个 `asar` 库文件代替 `app` 文件夹，你需要修改这个库文件的名字为 `app.asar` ， 然后将其放到 Electron 的资源文件夹下，然后 Electron 就会试图读取这个库文件并从中启动。 如下所示：

在 OS X 中:

```
electron/Electron.app/Contents/Resources/
└── app.asar
```

在 Windows 和 Linux 中:

```
electron/resources/
└── app.asar
```

更多的细节请见 Application packaging.

更换名称与下载二进制文件

在使用 Electron 打包你的应用程序之后，你可能需要在分发给用户之前修改打包的名字。

Windows

你可以将 `electron.exe` 改成任意你喜欢的名字，然后可以使用像 rcedit 编辑它的 icon 和其他信息。

OS X

你可以将 `Electron.app` 改成任意你喜欢的名字，然后你也需要修改这些文件中的 `CFBundleDisplayName`， `CFBundleIdentifier` 以及 `CFBundleName` 字段。 这些文件如下：

- `Electron.app/Contents/Info.plist`
- `Electron.app/Contents/Frameworks/Electron Helper.app/Contents/Info.plist`

你也可以重命名帮助应用程序以避免在应用程序监视器中显示 `Electron Helper`， 但是请确保你已经修改了帮助应用的可执行文件的名字。

一个改过名字的应用程序的构造可能是这样的：

```
MyApp.app/Contents
├── Info.plist
├── MacOS/
│   └── MyApp
└── Frameworks/
    ├── MyApp Helper EH.app
    |   ├── Info.plist
    |   └── MacOS/
    |       └── MyApp Helper EH
    ├── MyApp Helper NP.app
    |   ├── Info.plist
    |   └── MacOS/
    |       └── MyApp Helper NP
    └── MyApp Helper.app
        ├── Info.plist
        └── MacOS/
            └── MyApp Helper
```

Linux

你可以将 `electron` 改成任意你喜欢的名字。

通过重编译源代码来更换名称

通过修改产品名称并重编译源代码来更换 Electron 的名称也是可行的。 你需要修改 `atom.gyp` 文件并彻底重编译一次。

grunt打包脚本

手动检查 Electron 代码并重编译是很复杂晦涩的，因此有一个Grunt任务可以自动的处理 这些内容 grunt-build-atom-shell.

这个任务会自动的处理编辑 `.gyp` 文件，从源代码进行编译，然后重编译你的应用程序的本地 Node 模块以匹配这个新的可执行文件的名称。

Mac App Store 应用提交向导

自从 v0.34.0, Electron 就允许提交应用包到 Mac App Store (MAS) . 这个向导提供的信息有 : 如何提交应用和 MAS 构建的限制.

注意: 从 v0.36.0，当应用成为沙箱之后，会有一个 bug 阻止 GPU 进程开启 , 所以在这个 bug 修复之前，建议使用 v0.35.x .更多查看 issue #3871 .

注意: 提交应用到 Mac App Store 需要参加 Apple Developer Program , 这需要花钱.

如何提交

下面步骤介绍了一个简单的提交应用到商店方法.然而，这些步骤不能保证你的应用被 Apple 接受；你仍然需要阅读 Apple 的 Submitting Your App 关于如何满足 Mac App Store 要求的向导.

获得证书

为了提交应用到商店，首先需要从 Apple 获得一个证书.可以遵循 existing guides.

App 签名

获得证书之后，你可以使用 Application Distribution 打包你的应用, 然后前往提交你的应用.这个步骤基本上和其他程序一样，但是这 key 一个个的标识 Electron 的每个依赖.

首先，你需要准备2个授权文件 .

`child.plist` :

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyL
<plist version="1.0">
  <dict>
    <key>com.apple.security.app-sandbox</key>
    <true/>
    <key>com.apple.security.inherit</key>
    <true/>
  </dict>
</plist>
```

`parent.plist` :

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN" "http://www.apple.com/DTDs/PropertyL
<plist version="1.0">
  <dict>
    <key>com.apple.security.app-sandbox</key>
    <true/>
  </dict>
</plist>
```

然后使用下面的脚本标识你的应用：

```
#!/bin/bash

# Name of your app.
APP="YourApp"
# The path of you app to sign.
APP_PATH="/path/to/YouApp.app"
# The path to the location you want to put the signed package.
RESULT_PATH="~/Desktop/$APP.pkg"
# The name of certificates you requested.
APP_KEY="3rd Party Mac Developer Application: Company Name (APPIDENTITY)"
INSTALLER_KEY="3rd Party Mac Developer Installer: Company Name (APPIDENTITY)"

FRAMEWORKS_PATH="$APP_PATH/Contents/Frameworks"

codesign --deep -fs "$APP_KEY" --entitlements child.plist "$FRAMEWORKS_PATH/Electron Fram
codesign --deep -fs "$APP_KEY" --entitlements child.plist "$FRAMEWORKS_PATH/$APP Helper.a
codesign --deep -fs "$APP_KEY" --entitlements child.plist "$FRAMEWORKS_PATH/$APP Helper E
codesign --deep -fs "$APP_KEY" --entitlements child.plist "$FRAMEWORKS_PATH/$APP Helper N
if [ -d "$FRAMEWORKS_PATH/Squirrel.framework/Versions/A" ]; then
  # Signing a non-MAS build.
  codesign --deep -fs "$APP_KEY" --entitlements child.plist "$FRAMEWORKS_PATH/Mantle.fram
  codesign --deep -fs "$APP_KEY" --entitlements child.plist "$FRAMEWORKS_PATH/ReactiveCoc
  codesign --deep -fs "$APP_KEY" --entitlements child.plist "$FRAMEWORKS_PATH/Squirrel.fr
fi
codesign -fs "$APP_KEY" --entitlements parent.plist "$APP_PATH"

productbuild --component "$APP_PATH" /Applications --sign "$INSTALLER_KEY" "$RESULT_PATH"
```

如果你是 OS X 下的应用沙箱使用新手，应当仔细阅读 Apple 的 Enabling App Sandbox 来有一点基础,然后向授权文件添加你的应用需要的许可 keys .

上传你的应用并检查提交

在签名应用之后，可以使用应用 Loader 来上传到 iTunes 链接处理 , 确保在上传之前你已经 created a record. 然后你能 submit your app for review.

MAS构建限制

为了让你的应用沙箱满足所有条件，在 MAS 构建的时候，下面的模块被禁用了：

- `crashReporter`
- `autoUpdater`

并且下面的行为也改变了:

- 一些机子的视频采集功能无效.
- 某些特征不可访问.
- Apps 不可识别 DNS 改变.

也由于应用沙箱的使用方法，应用可以访问的资源被严格限制了；阅读更多信息 App Sandboxing .

Electron 使用的加密算法

取决于你所在地方的国家和地区，Mac App Store 或许需要记录你应用的加密算法，甚至要求你提交一个 U.S 加密注册(ERN) 许可的复印件.

Electron 使用下列加密算法:

- AES - NIST SP 800-38A, NIST SP 800-38D, RFC 3394
- HMAC - FIPS 198-1
- ECDSA - ANS X9.62–2005
- ECDH - ANS X9.63–2001
- HKDF - NIST SP 800-56C
- PBKDF2 - RFC 2898
- RSA - RFC 3447
- SHA - FIPS 180-4
- Blowfish - https://www.schneier.com/cryptography/blowfish/
- CAST - RFC 2144, RFC 2612
- DES - FIPS 46-3
- DH - RFC 2631
- DSA - ANSI X9.30
- EC - SEC 1
- IDEA - "On the Design and Security of Block Ciphers" book by X. Lai
- MD2 - RFC 1319
- MD4 - RFC 6150
- MD5 - RFC 1321
- MDC2 - ISO/IEC 10118-2
- RC2 - RFC 2268
- RC4 - RFC 4345
- RC5 - http://people.csail.mit.edu/rivest/Rivest-rc5rev.pdf
- RIPEMD - ISO/IEC 10118-3

如何获取 ERN 许可, 可看这篇文章: How to legally submit an app to Apple's App Store when it uses encryption (or how to obtain an ERN).

应用打包

为舒缓 Windows 下路径名过长的问题issues， 也略对 `require` 加速以及简单隐匿你的源代码，你可以通过极小的源代码改动将你的应用打包成 asar。

生成 `asar` 包

asar 是一种将多个文件合并成一个文件的类 tar 风格的归档格式。 Electron 可以无需解压，即从其中读取任意文件内容。

参照如下步骤将你的应用打包成 `asar`：

1. 安装 asar

```
$ npm install -g asar
```

2. 用 `asar pack` 打包

```
$ asar pack your-app app.asar
```

使用 `asar` 包

在 Electron 中有两类 APIs：Node.js 提供的 Node API 和 Chromium 提供的 Web API。 这两种 API 都支持从 `asar` 包中读取文件。

Node API

由于 Electron 中打了特别补丁， Node API 中如 `fs.readFile` 或者 `require` 之类 的方法可以将 `asar` 视之为虚拟文件夹，读取 `asar` 里面的文件就和从真实的文件系统中读取一样。

例如，假设我们在 `/path/to` 文件夹下有个 `example.asar` 包：

```
$ asar list /path/to/example.asar
/app.js
/file.txt
/dir/module.js
/static/index.html
/static/main.css
/static/jquery.min.js
```

从 `asar` 包读取一个文件：

```
const fs = require('fs');
fs.readFileSync('/path/to/example.asar/file.txt');
```

列出 `asar` 包中根目录下的所有文件：

```
const fs = require('fs');
fs.readdirSync('/path/to/example.asar');
```

使用 `asar` 包中的一个模块：

```
require('/path/to/example.asar/dir/module.js');
```

你也可以使用 `BrowserWindow` 来显示一个 `asar` 包里的 web 页面：

```
const BrowserWindow = require('electron').BrowserWindow;
var win = new BrowserWindow({width: 800, height: 600});
win.loadURL('file:///path/to/example.asar/static/index.html');
```

Web API

在 Web 页面里，用 `file:` 协议可以获取 `asar` 包中文件。和 Node API 一样，视 `asar` 包如虚拟文件夹。

例如，用 `$.get` 获取文件:

```
<script>
var $ = require('./jquery.min.js');
$.get('file:///path/to/example.asar/file.txt', function(data) {
  console.log(data);
});
</script>
```

像“文件”那样处理 `asar` 包

有些场景，如：核查 `asar` 包的校验和，我们需要像读取“文件”那样读取 `asar` 包的内容(而不是当成虚拟文件夹)。你可以使用内置的 `original-fs` （提供和 `fs` 一样的 API）模块来读取 `asar` 包的真实信息。

```
var originalFs = require('original-fs');
originalFs.readFileSync('/path/to/example.asar');
```

Node API 缺陷

尽管我们已经尽了最大努力使得 `asar` 包在 Node API 下的应用尽可能的趋向于真实的目录结构，但仍有一些底层 Node API 我们无法保证其正常工作。

`asar` 包是只读的

`asar` 包中的内容不可更改，所以 Node APIs 里那些可以用来修改文件的方法在对待 `asar` 包时都无法正常工作。

Working Directory 在 `asar` 包中无效

尽管 `asar` 包是虚拟文件夹，但其实并没有真实的目录架构对应在文件系统里，所以你不可能将 working Directory 设置成 `asar` 包里的一个文件夹。将 `asar` 中的文件夹以 `cwd` 形式作为参数传入一些 API 中也会报错。

API 中的额外“开箱”

大部分 `fs` API 可以无需解压即从 `asar` 包中读取文件或者文件的信息，但是在处理一些依赖真实文件路径的底层 系统方法时，Electron 会将所需文件解压到临时目录下，然后将临时目录下的真实文件路径传给底层系统方法使其正 常工作。 对于这类API，耗费会略多一些。

以下是一些需要额外解压的 API：

- `child_process.execFile`
- `child_process.execFileSync`
- `fs.open`
- `fs.openSync`
- `process.dlopen` - `require` native模块时用到

`fs.stat` 获取的 stat 信息不可靠

对 `asar` 包中的文件取 `fs.stat`，返回的 `Stats` 对象不是精确值，因为这些文件不是真实存在于文件系 统里。所以除了文件大小和文件类型以外，你不应该依赖 `Stats` 对象的值。

执行 `asar` 包中的程序

Node 中有一些可以执行程序的 API，如 `child_process.exec`，`child_process.spawn` 和 `child_process.execFile` 等，但只有 `execFile` 可以执行 `asar` 包中的程序。

因为 `exec` 和 `spawn` 允许 `command` 替代 `file` 作为输入，而 `command` 是需要在 shell 下执行的，目前没有 可靠的方法来判断 `command` 中是否在操作一个 `asar` 包中的文件，而且即便可以判断，我们依旧无法保证可以在无任何 副作用的情况下替换 `command` 中的文件路径。

打包时排除文件

如上所述，一些 Node API 会在调用时将文件解压到文件系统中，除了效率问题外，也有可能引起杀毒软件的注意！

为解决这个问题，你可以在生成 `asar` 包时使用 `--unpack` 选项来排除一些文件，使其不打包到 `asar` 包中， 下面是如何排除一些用作共享用途的 native 模块的方法：

```
$ asar pack app app.asar --unpack *.node
```

经过上述命令后，除了生成的 `app.asar` 包以外，还有一个包含了排除文件的 `app.asar.unpacked` 文件夹， 你需要将这个文件夹一起拷贝，提供给用户。

使用原生模块

Electron 同样也支持原生模块，但由于和官方的 Node 相比使用了不同的 V8 引擎，如果你想编译原生模块，则需要手动设置 Electron 的 headers 的位置。

原生Node模块的兼容性

当 Node 开始换新的V8引擎版本时，原生模块可能“坏”掉。为确保一切工作正常，你需要检查你想要使用的原生模块是否被 Electron 内置的 Node 支持。你可以在这里查看 Electron 内置的 Node 版本，或者使用 `process.version` (参考：快速入门)查看。

考虑到 NAN 可以使你的开发更容易对多版本 Node 的支持，建议使用它来开发你自己的模块。你也可以使用 NAN 来移植旧的模块到新的 Nod e版本，以使它们可以在新的 Electron 下良好工作。

如何安装原生模块

如下三种方法教你安装原生模块：

最简单方式

最简单的方式就是通过 `electron-rebuild` 包重新编译原生模块，它帮你自动完成了下载 headers、编译原生模块等步骤：

```
npm install --save-dev electron-rebuild

# 每次运行"npm install"时，也运行这条命令
./node_modules/.bin/electron-rebuild

# 在windows下如果上述命令遇到了问题，尝试这个：
.\node_modules\.bin\electron-rebuild.cmd
```

通过 npm 安装

你当然也可以通过 `npm` 安装原生模块。大部分步骤和安装普通模块时一样，除了以下一些系统环境变量你需要自己操作：

```
export npm_config_disturl=https://atom.io/download/atom-shell
export npm_config_target=0.33.1
export npm_config_arch=x64
export npm_config_runtime=electron
HOME=~/.electron-gyp npm install module-name
```

通过 node-gyp 安装

你需要告诉 `node-gyp` 去哪下载 Electron 的 headers，以及下载什么版本：

```
$ cd /path-to-module/
$ HOME=~/.electron-gyp node-gyp rebuild --target=0.29.1 --arch=x64 --dist-url=https://ato
```

`HOME=~/.electron-gyp` 设置去哪找开发时的 headers。

`--target=0.29.1` 设置了 Electron 的版本

`--dist-url=...` 设置了 Electron 的 headers 的下载地址

`--arch=x64` 设置了该模块为适配64位操作系统而编译

主进程调试

浏览器窗口的开发工具仅能调试渲染器的进程脚本（比如 web 页面）。为了提供一个可以调试主进程 的方法，Electron 提供了 `--debug` 和 `--debug-brk` 开关。

命令行开关

使用如下的命令行开关来调试 Electron 的主进程：

`--debug=[port]`

当这个开关用于 Electron 时，它将会监听 V8 引擎中有关 `port` 的调试器协议信息。 默认的 `port` 是 `5858`。

`--debug-brk=[port]`

就像 `--debug` 一样，但是会在第一行暂停脚本运行。

使用 node-inspector 来调试

备注： Electron 目前对 node-inspector 支持的不是特别好， 如果你通过 node-inspector 的 console 来检查 `process` 对象，主进程就会崩溃。

1. 确认你已经安装了 node-gyp 所需工具

2. 安装 node-inspector

```
$ npm install node-inspector
```

3. 安装 `node-pre-gyp` 的一个修订版

```
$ npm install git+https://git@github.com/enlight/node-pre-gyp.git#detect-electron-runtime
```

4. 为 Electron 重新编译 `node-inspector` `v8` 模块（将 target 参数修改为你的 Electron 的版本号）

```
$ node_modules/.bin/node-pre-gyp --target=0.36.2 --runtime=electron --fallback-to-build -
$ node_modules/.bin/node-pre-gyp --target=0.36.2 --runtime=electron --fallback-to-build -
```

[How to install native modules][how-to-install-native-modules].

5. 打开 Electron 的调试模式

你也可以用调试参数来运行 Electron ：

```
$ electron --debug=5858 your/app
```

或者，在第一行暂停你的脚本：

```
$ electron --debug-brk=5858 your/app
```

6. 使用 Electron 开启 node-inspector 服务

```
$ ELECTRON_RUN_AS_NODE=true path/to/electron.exe node_modules/node-inspector/bin/inspecto
```

7. 加载调试器界面

在 Chrome 中打开 http://127.0.0.1:8080/debug?ws=127.0.0.1:8080&port=5858

使用 Selenium 和 WebDriver

引自ChromeDriver - WebDriver for Chrome:

> WebDriver 是一款开源的支持多浏览器的自动化测试工具。它提供了操作网页、用户输入、JavaScript 执行等能力。ChromeDriver 是一个实现了 WebDriver 与 Chromium 联接协议的独立服务。它也是由开发了 Chromium 和 WebDriver 的团队开发的。

为了能够使 `chromedriver` 和 Electron 一起正常工作，我们需要告诉它 Electron 在哪，并且让它相信 Electron 就是 Chrome 浏览器。

通过 WebDriverJs 配置

WebDriverJs 是一个可以配合 WebDriver 做测试的 node 模块，我们会用它来做个演示。

1. 启动 ChromeDriver

首先，你要下载 `chromedriver` ，然后运行以下命令：

```
$ ./chromedriver
Starting ChromeDriver (v2.10.291558) on port 9515
Only local connections are allowed.
```

记住 `9515` 这个端口号，我们后面会用到

2. 安装 WebDriverJS

```
$ npm install selenium-webdriver
```

3. 联接到 ChromeDriver

在 Electron 下使用 `selenium-webdriver` 和其平时的用法并没有大的差异，只是你需要手动设置连接 ChromeDriver，以及 Electron 的路径：

```
const webdriver = require('selenium-webdriver');

var driver = new webdriver.Builder()
  // "9515" 是ChromeDriver使用的端口
  .usingServer('http://localhost:9515')
  .withCapabilities({
    chromeOptions: {
      // 这里设置Electron的路径
      binary: '/Path-to-Your-App.app/Contents/MacOS/Atom',
    }
  })
  .forBrowser('electron')
  .build();

driver.get('http://www.google.com');
driver.findElement(webdriver.By.name('q')).sendKeys('webdriver');
driver.findElement(webdriver.By.name('btnG')).click();
driver.wait(function() {
 return driver.getTitle().then(function(title) {
   return title === 'webdriver - Google Search';
 });
}, 1000);

driver.quit();
```

通过 WebdriverIO 配置

WebdriverIO 也是一个配合 WebDriver 用来测试的 node 模块

1. 启动 ChromeDriver

首先，下载 `chromedriver` ，然后运行以下命令：

```
$ chromedriver --url-base=wd/hub --port=9515
Starting ChromeDriver (v2.10.291558) on port 9515
Only local connections are allowed.
```

记住 `9515` 端口，后面会用到

2. 安装 WebdriverIO

```
$ npm install webdriverio
```

3. 连接到 ChromeDriver

```
const webdriverio = require('webdriverio');
var options = {
    host: "localhost", // 使用localhost作为ChromeDriver服务器
    port: 9515,         // "9515"是ChromeDriver使用的端口
    desiredCapabilities: {
        browserName: 'chrome',
        chromeOptions: {
          binary: '/Path-to-Your-App/electron', // Electron的路径
          args: [/* cli arguments */]           // 可选参数，类似：'app=' + /path/to/your/ap
        }
    }
};

var client = webdriverio.remote(options);

client
    .init()
    .url('http://google.com')
    .setValue('#q', 'webdriverio')
    .click('#btnG')
    .getTitle().then(function(title) {
        console.log('Title was: ' + title);
    })
    .end();
```

工作流程

无需重新编译 Electron，只要把 app 的源码放到 Electron的资源目录 里就可直接开始测试了。

当然，你也可以在运行 Electron 时传入参数指定你 app 的所在文件夹。这步可以免去你拷贝一粘贴你的 app 到 Electron 的资源目录。

DevTools扩展

为了使调试更容易，Electron 原生支持 Chrome DevTools Extension。

对于大多数DevTools的扩展，你可以直接下载源码，然后通过 `BrowserWindow.addDevToolsExtension` API 加载它们。Electron会记住已经加载了哪些扩展，所以你不需要每次创建一个新window时都调用 `BrowserWindow.addDevToolsExtension` API。

注：**React DevTools**目前不能直接工作，详情留意 **https://github.com/electron/electron/issues/915**

例如，要用React DevTools Extension，你得先下载他的源码：

```
$ cd /some-directory
$ git clone --recursive https://github.com/facebook/react-devtools.git
```

参考 `react-devtools/shells/chrome/Readme.md` 来编译这个扩展源码。

然后你就可以在任意页面的 DevTools 里加载 React DevTools 了，通过控制台输入如下命令加载扩展：

```
const BrowserWindow = require('electron').remote.BrowserWindow;
BrowserWindow.addDevToolsExtension('/some-directory/react-devtools/shells/chrome');
```

要卸载扩展，可以调用 `BrowserWindow.removeDevToolsExtension` API (扩展名作为参数传入)，该扩展在下次打开DevTools时就不会加载了：

```
BrowserWindow.removeDevToolsExtension('React Developer Tools');
```

DevTools 扩展的格式

理论上，Electron 可以加载所有为 chrome 浏览器编写的 DevTools 扩展，但它们必须存放在文件夹里。那些以 `crx` 形式发布的扩展是不能被加载的，除非你把它们解压到一个文件夹里。

后台运行(background pages)

Electron 目前并不支持 chrome 扩展里的后台运行(background pages)功能，所以那些依赖此特性的 DevTools 扩展在 Electron 里可能无法正常工作。

`chrome.*` APIs

有些 chrome 扩展使用了 `chrome.*` APIs，而且这些扩展在 Electron 中需要额外实现一些代码才能使用，所以并不是所有的这类扩展都已经在 Electron 中实现完毕了。

考虑到并非所有的 `chrome.*` APIs 都实现完毕，如果 DevTools 正在使用除了 `chrome.devtools.*` 之外的其它 APIs，这个扩展很可能无法正常工作。你可以通过报告这个扩展的异常信息，这样做方便我们对该扩展的支持。

使用 Pepper Flash 插件

Electron 现在支持 Pepper Flash 插件。要在 Electron 里面使用 Pepper Flash 插件，你需 要手动设置 Pepper Flash 的路径和在你的应用里启用 Pepper Flash。

保留一份 Flash 插件的副本

在 OS X 和 Linux 上，你可以在 Chrome 浏览器的 `chrome://plugins` 页面上找到 Pepper Flash 的插件信息。插件的路径和版本会对 Election 对其的支持有帮助。你也可以把插件 复制到另一个路径以保留一份副本。

添加插件在 Electron 里的开关

你可以直接在命令行中用 `--ppapi-flash-path` 和 `ppapi-flash-version` 或者 在 app 的准备事件前调用 `app.commandLine.appendSwitch` 这个 method。同时， 添加 `browser-window` 的插件开关。 例如：

```
// Specify flash path. 设置 flash 路径
// On Windows, it might be /path/to/pepflashplayer.dll
// On OS X, /path/to/PepperFlashPlayer.plugin
// On Linux, /path/to/libpepflashplayer.so
app.commandLine.appendSwitch('ppapi-flash-path', '/path/to/libpepflashplayer.so');

// Specify flash version, for example, v17.0.0.169 设置版本号
app.commandLine.appendSwitch('ppapi-flash-version', '17.0.0.169');

app.on('ready', function() {
  mainWindow = new BrowserWindow({
    'width': 800,
    'height': 600,
    'web-preferences': {
      'plugins': true
    }
  });
  mainWindow.loadURL('file://' + __dirname + '/index.html');
  // Something else
});
```

使用 `<webview>` 标签启用插件

在 `<webview>` 标签里添加 `plugins` 属性。

```
<webview src="http://www.adobe.com/software/flash/about/" plugins></webview>
```

使用 Widevine CDM 插件

在 Electron ，你可以使用 Widevine CDM 插件装载 Chrome 浏览器 .

获取插件

Electron 没有为 Widevine CDM 插件 配制许可 reasons, 为了获得它，首先需要安装官方的 chrome 浏览器，这匹配了体系架构和 Electron 构建使用的 chrome 版本 .

注意: Chrome 浏览器的主要版本必须和 Electron 使用的版本一样，否则插件不会有效，虽然 `navigator.plugins` 会显示你已经安装了它 .

Windows & OS X

在 Chrome 浏览器中打开 `chrome://components/` ，找到 `WidevineCdm` 并且确定它更新到最新版本，然后你可以从 `APP_DATA/Google/Chrome/WidevineCDM/VERSION/_platform_specific/PLATFORM_ARCH/` 路径找到所有的插件二进制文件 .

`APP_DATA` 是系统存放数据的地方，在 Windows 上它是 `%LOCALAPPDATA%` , 在 OS X 上它是 `~/Library/Application Support` . `VERSION` 是 Widevine CDM 插件的版本字符串, 类似 `1.4.8.866` . `PLATFORM` 是 `mac` 或 `win` . `ARCH` 是 `x86` 或 `x64` .

在 Windows，必要的二进制文件是 `widevinecdm.dll` and `widevinecdmadapter.dll` , 在 OS X ，它们是 `libwidevinecdm.dylib` 和 `widevinecdmadapter.plugin` . 你可以将它们复制到任何你喜欢的地方，但是它们必须要放在一起.

Linux

在 Linux ，Chrome 浏览器将插件的二进制文件装载在一起 , 你可以在 `/opt/google/chrome` 下找到,文件名是 `libwidevinecdm.so` 和 `libwidevinecdmadapter.so` .

使用插件

在获得了插件文件后，你可以使用 `--widevine-cdm-path` 命令行开关来将 `widevinecdmadapter` 的路径传递给 Electron , 插件版本使用 `--widevine-cdm-version` 开关.

注意: 虽然只有 `widevinecdmadapter` 的二进制文件传递给了 Electron, `widevinecdm` 二进制文件应当放在它的旁边.

必须在 `app` 模块的 `ready` 事件触发之前使用命令行开关，并且 page 使用的插件必须激活.

示例代码：

```
// You have to pass the filename of `widevinecdmadapter` here, it is
// * `widevinecdmadapter.plugin` on OS X,
// * `libwidevinecdmadapter.so` on Linux,
// * `widevinecdmadapter.dll` on Windows.
app.commandLine.appendSwitch('widevine-cdm-path', '/path/to/widevinecdmadapter.plugin');
// The version of plugin can be got from `chrome://plugins` page in Chrome.
app.commandLine.appendSwitch('widevine-cdm-version', '1.4.8.866');

var mainWindow = null;
app.on('ready', function() {
  mainWindow = new BrowserWindow({
    webPreferences: {
      // The `plugins` have to be enabled.
      plugins: true
    }
  })
});
```

验证插件

为了验证插件是否工作，你可以使用下面的方法：

- 打开开发者工具查看是否 `navigator.plugins` 包含了 Widevine CDM 插件.
- 打开 `https://shaka-player-demo.appspot.com/` 加载一个使用 `Widevine` 的 manifest.
- 打开 http://www.dash-player.com/demo/drm-test-area/, 检查是否界面输出 `bitdash uses Widevine in your browser`, 然后播放 video.

教程

快速入门

简介

Electron 可以让你使用纯 JavaScript 调用丰富的原生 APIs 来创造桌面应用。你可以把它看作是专注于桌面应用而不是 web 服务器的，io.js 的一个变体。

这不意味着 Electron 是绑定了 GUI 库的 JavaScript。相反，Electron 使用 web 页面作为它的 GUI，所以你能把它看作成一个被 JavaScript 控制的，精简版的 Chromium 浏览器。

主进程

在 Electron 里，运行 `package.json` 里 `main` 脚本的进程被称为主进程。在主进程运行的脚本可以以创建 web 页面的形式展示 GUI。

渲染进程

由于 Electron 使用 Chromium 来展示页面，所以 Chromium 的多进程结构也被充分利用。每个 Electron 的页面都在运行着自己的进程，这样的进程我们称之为渲染进程。

在一般浏览器中，网页通常会在沙盒环境下运行，并且不允许访问原生资源。然而，Electron 用户拥有在网页中调用 io.js 的 APIs 的能力，可以与底层操作系统直接交互。

主进程与渲染进程的区别

主进程使用 BrowserWindow 实例创建网页。每个 BrowserWindow 实例都在自己的渲染进程里运行着一个网页。当一个 BrowserWindow 实例被销毁后，相应的渲染进程也会被终止。

主进程管理所有页面和与之对应的渲染进程。每个渲染进程都是相互独立的，并且只关心他们自己的网页。

由于在网页里管理原生 GUI 资源是非常危险而且容易造成资源泄露，所以在网页面调用 GUI 相关的 APIs 是不被允许的。如果你想在网页里使用 GUI 操作，其对应的渲染进程必须与主进程进行通讯，请求主进程进行相关的 GUI 操作。

在 Electron，我们提供用于在主进程与渲染进程之间通讯的 ipc 模块。并且也有一个远程进程调用风格的通讯模块 remote。

打造你第一个 Electron 应用

大体上，一个 Electron 应用的目录结构如下：

```
your-app/
├── package.json
├── main.js
└── index.html
```

`package.json` 的格式和 Node 的完全一致，并且那个被 `main` 字段声明的脚本文件是你的应用的启动脚本，它运行在主进程上。你应用里的 `package.json` 看起来应该像：

```
{
  "name"    : "your-app",
  "version" : "0.1.0",
  "main"    : "main.js"
}
```

注意：如果 `main` 字段没有在 `package.json` 声明，Electron会优先加载 `index.js` 。

`main.js` 应该用于创建窗口和处理系统时间，一个典型的例子如下：

```
var app = require('app');  // 控制应用生命周期的模块。
var BrowserWindow = require('browser-window');  // 创建原生浏览器窗口的模块

// 保持一个对于 window 对象的全局引用，不然，当 JavaScript 被 GC，
// window 会被自动地关闭
var mainWindow = null;

// 当所有窗口被关闭了，退出。
app.on('window-all-closed', function() {
  // 在 OS X 上，通常用户在明确地按下 Cmd + Q 之前
  // 应用会保持活动状态
  if (process.platform != 'darwin') {
    app.quit();
  }
});

// 当 Electron 完成了初始化并且准备创建浏览器窗口的时候
// 这个方法就被调用
app.on('ready', function() {
  // 创建浏览器窗口。
  mainWindow = new BrowserWindow({width: 800, height: 600});

  // 加载应用的 index.html
  mainWindow.loadURL('file://' + __dirname + '/index.html');

  // 打开开发工具
  mainWindow.openDevTools();

  // 当 window 被关闭，这个事件会被发出
  mainWindow.on('closed', function() {
    // 取消引用 window 对象，如果你的应用支持多窗口的话，
    // 通常会把多个 window 对象存放在一个数组里面，
    // 但这次不是。
    mainWindow = null;
  });
});
```

最后，你想展示的 `index.html` ：

```
<!DOCTYPE html>
<html>
  <head>
    <title>Hello World!</title>
  </head>
  <body>
    <h1>Hello World!</h1>
    We are using io.js <script>document.write(process.version)</script>
    and Electron <script>document.write(process.versions['electron'])</script>.
  </body>
</html>
```

运行你的应用

一旦你创建了最初的 `main.js` ， `index.html` 和 `package.json` 这几个文件，你可能会想尝试在本地运行并测试，看看是不是和期望的那样正常运行。

electron-prebuild

如果你已经用 `npm` 全局安装了 `electron-prebuilt` ，你只需要按照如下方式直接运行你的应用：

```
electron .
```

如果你是局部安装，那运行：

```
./node_modules/.bin/electron .
```

手工下载 Electron 二进制文件

如果你手工下载了 Electron 的二进制文件，你也可以直接使用其中的二进制文件直接运行你的应用。

Windows

```
$ .\electron\electron.exe your-app\
```

Linux

```
$ ./electron/electron your-app/
```

OS X

```
$ ./Electron.app/Contents/MacOS/Electron your-app/
```

`Electron.app` 里面是 Electron 发布包，你可以在这里下载到。

以发行版本运行

在你完成了你的应用后，你可以按照应用部署指导发布一个版本，并且以已经打包好的形式运行应用。

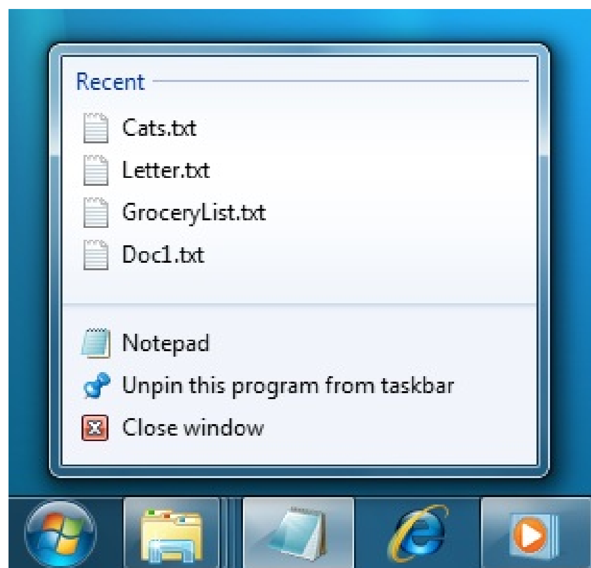
桌面环境集成

不同的操作系统在各自的桌面应用上提供了不同的特性。例如，在 windows 上应用曾经打开的文件会出现在任务栏的跳转列表，在 Mac 上，应用可以把自定义菜单放在鱼眼菜单上。

本章将会说明怎样使用 Electron APIs 把你的应用和桌面环境集成到一块。

最近文档 (Windows & OS X)

Windows 和 OS X 提供获取最近文档列表的便捷方式，那就是打开跳转列表或者鱼眼菜单。

跳转列表：

鱼眼菜单：



为了增加一个文件到最近文件列表，你可以使用 app.addRecentDocument API:

```
var app = require('app');
app.addRecentDocument('/Users/USERNAME/Desktop/work.type');
```

或者你也可以使用 app.clearRecentDocuments API 来清空最近文件列表。

```
app.clearRecentDocuments();
```

Windows 需注意

为了这个特性在 Windows 上表现正常，你的应用需要被注册成为一种文件类型的句柄，否则，在你注册之前，文件不会出现在跳转列表。你可以在 Application Registration 里找到任何关于注册事宜的说明。

OS X 需注意

当一个文件被最近文件列表请求时，`app` 模块里的 `open-file` 事件将会被发出。

自定义的鱼眼菜单(OS X)

OS X 可以让开发者定制自己的菜单，通常会包含一些常用特性的快捷方式。

菜单中的终端



使用 `app.dock.setMenu` API 来设置你的菜单，这仅在 OS X 上可行：

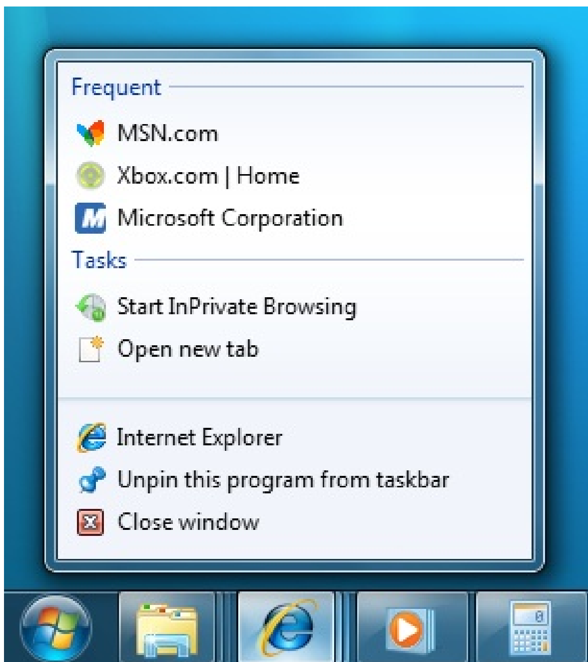
```
var app = require('app');
var Menu = require('menu');
var dockMenu = Menu.buildFromTemplate([
  { label: 'New Window', click: function() { console.log('New Window'); } },
  { label: 'New Window with Settings', submenu: [
    { label: 'Basic' },
    { label: 'Pro'}
  ]},
  { label: 'New Command...'}
]);
app.dock.setMenu(dockMenu);
```

用户任务(Windows)

在 Windows，你可以特别定义跳转列表的 `Tasks` 目录的行为，引用 MSDN 的文档：

> Applications define tasks based on both the program's features and the key things a user is expected to do with them. Tasks should be context-free, in that the application does not need to be running for them to work. They should also be the statistically most common actions that a normal user would perform in an application, such as compose an email message or open the calendar in a mail program, create a new document in a word processor, launch an application in a certain mode, or launch one of its subcommands. An application should not clutter the menu with advanced features that standard users won't need or one-time actions such as registration. Do not use tasks for promotional items such as upgrades or special offers.
>
> It is strongly recommended that the task list be static. It should remain the same regardless of the state or status of the application. While it is possible to vary the list dynamically, you should consider that this could confuse the user who does not expect that portion of the destination list to change.

IE 的任务



不同于 OS X 的鱼眼菜单，Windows 上的用户任务表现得更像一个快捷方式，比如当用户点击一个任务，一个程序将会被传入特定的参数并且运行。

你可以使用 app.setUserTasks API 来设置你的应用中的用户任务：

```
var app = require('app');
app.setUserTasks([
  {
    program: process.execPath,
    arguments: '--new-window',
    iconPath: process.execPath,
    iconIndex: 0,
    title: 'New Window',
    description: 'Create a new window'
  }
]);
```

调用 `app.setUserTasks` 并传入空数组就可以清除你的任务列表：

```
app.setUserTasks([]);
```

当你的应用关闭时，用户任务会仍然会出现，在你的应用被卸载前，任务指定的图标和程序的路径必须是存在的。

缩略图工具栏

在 Windows，你可以在任务栏上添加一个按钮来当作应用的缩略图工具栏。它将提供用户一种用户访问常用窗口的方式，并且不需要恢复或者激活窗口。

在 MSDN，它被如是说：

> This toolbar is simply the familiar standard toolbar common control. It has a maximum of seven buttons. Each button's ID, image, tooltip, and state are defined in a structure, which is then passed to the taskbar. The application can show, enable, disable, or hide buttons from the thumbnail toolbar as required by its current state.
>
> For example, Windows Media Player might offer standard media transport controls such as play, pause, mute, and stop.

Windows Media Player 的缩略图工具栏



你可以使用 BrowserWindow.setThumbarButtons 来设置你的应用的缩略图工具栏。

```
var BrowserWindow = require('browser-window');
var path = require('path');
var win = new BrowserWindow({
  width: 800,
  height: 600
});
win.setThumbarButtons([
  {
    tooltip: "button1",
    icon: path.join(__dirname, 'button1.png'),
    click: function() { console.log("button2 clicked"); }
  },
  {
    tooltip: "button2",
    icon: path.join(__dirname, 'button2.png'),
    flags:['enabled', 'dismissonclick'],
    click: function() { console.log("button2 clicked."); }
  }
]);
```

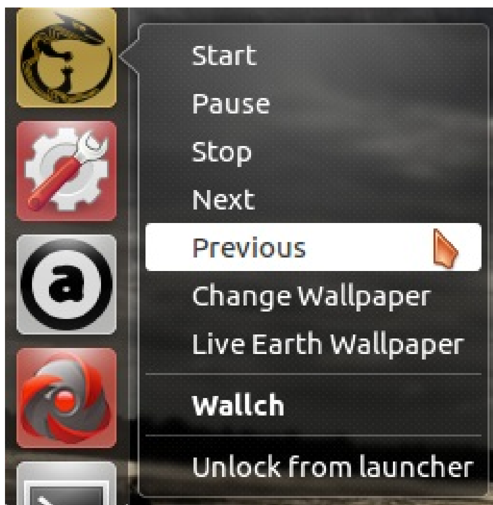
调用 `BrowserWindow.setThumbarButtons` 并传入空数组即可清空缩略图工具栏：

```
win.setThumbarButtons([]);
```

Unity launcher 快捷方式(Linux)

在 Unity,你可以通过改变 `.desktop` 文件来增加自定义运行器的快捷方式，详情看 Adding shortcuts to a launcher。

Audacious 运行器的快捷方式：

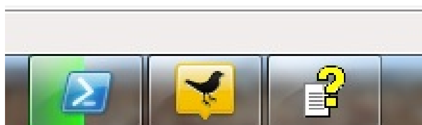


任务栏的进度条(Windows & Unity)

在 Windows，进度条可以出现在一个任务栏按钮之上。这可以提供进度信息给用户而不需要用户切换应用窗口。

Unity DE 也具有同样的特性，在运行器上显示进度条。

在任务栏上的进度条：



在 Unity 运行器上的进度条

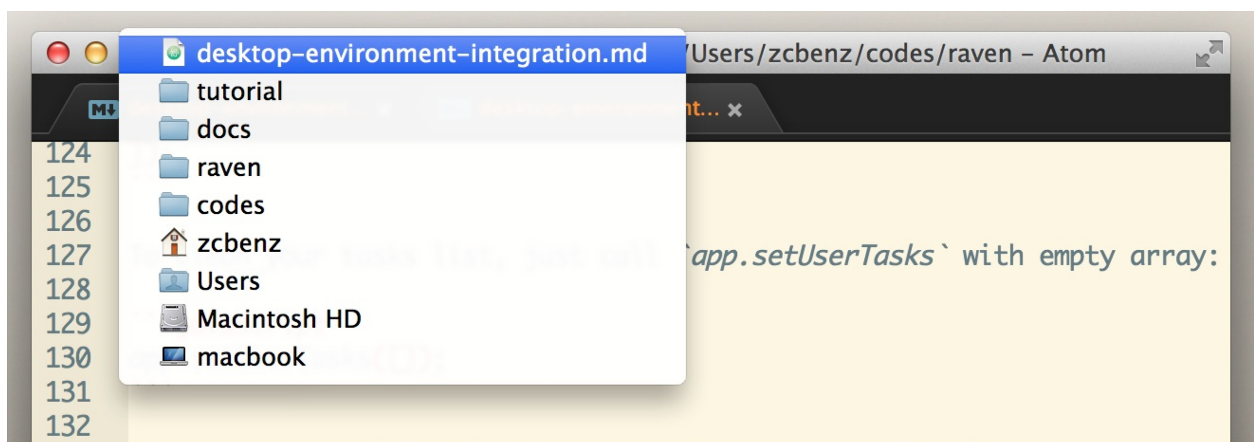


给一个窗口设置进度条，你可以调用 BrowserWindow.setProgressBar API：

```
var window = new BrowserWindow({...});
window.setProgressBar(0.5);
```

在 OS X，一个窗口可以设置它展示的文件，文件的图标可以出现在标题栏，当用户 Command-Click 或者 Control-Click 标题栏，文件路径弹窗将会出现。

展示文件弹窗菜单：



你可以调用 BrowserWindow.setRepresentedFilename 和 BrowserWindow.setDocumentEdited APIs：

```
var window = new BrowserWindow({...});
window.setRepresentedFilename('/etc/passwd');
window.setDocumentEdited(true);
```

在线/离线事件探测

使用标准 HTML5 APIs 可以实现在线和离线事件的探测，就像以下例子：

main.js

```
var app = require('app');
var BrowserWindow = require('browser-window');
var onlineStatusWindow;

app.on('ready', function() {
  onlineStatusWindow = new BrowserWindow({ width: 0, height: 0, show: false });
  onlineStatusWindow.loadURL('file://' + __dirname + '/online-status.html');
});
```

online-status.html

```
<!DOCTYPE html>
<html>
  <body>
    <script>
      var alertOnlineStatus = function() {
        window.alert(navigator.onLine ? 'online' : 'offline');
      };

      window.addEventListener('online',  alertOnlineStatus);
      window.addEventListener('offline',  alertOnlineStatus);

      alertOnlineStatus();
    </script>
  </body>
</html>
```

也会有人想要在主进程也有回应这些事件的实例。然后主进程没有 `navigator` 对象因此不能直接探测在线还是离线。使用 Electron 的进程间通讯工具，事件就可以在主进程被使，就像下面的例子：

main.js

```
var app = require('app');
var ipc = require('ipc');
var BrowserWindow = require('browser-window');
var onlineStatusWindow;

app.on('ready', function() {
  onlineStatusWindow = new BrowserWindow({ width: 0, height: 0, show: false });
  onlineStatusWindow.loadURL('file://' + __dirname + '/online-status.html');
});

ipc.on('online-status-changed', function(event, status) {
  console.log(status);
});
```

online-status.html

```
<!DOCTYPE html>
<html>
  <body>
    <script>
      var ipc = require('ipc');
      var updateOnlineStatus = function() {
        ipc.send('online-status-changed', navigator.onLine ? 'online' : 'offline');
      };

      window.addEventListener('online',  updateOnlineStatus);
      window.addEventListener('offline',  updateOnlineStatus);

      updateOnlineStatus();
    </script>
  </body>
</html>
```

API文档

简介

所有的Node.js's built-in modules在Electron中都可用，并且所有的node的第三方组件也可以放心使用（包括自身的模块）。

Electron也提供了一些额外的内置组件来开发传统桌面应用。一些组件只可以在主进程中使用，一些只可以在渲染进程中使用，但是也有部分可以在这2种进程中都可使用。

基本规则：GUI模块或者系统底层的模块只可以在主进程中使用。要使用这些模块，你应当很熟悉主进程vs渲染进程脚本的概念。

主进程脚本看起来像个普通的nodejs脚本

```
const electron = require('electron');
const app = electron.app;
const BrowserWindow = electron.BrowserWindow;

var window = null;

app.on('ready', function() {
  window = new BrowserWindow({width: 800, height: 600});
  window.loadURL('https://github.com');
});
```

渲染进程和传统的web界面一样，除了它具有使用node模块的能力：

```
<!DOCTYPE html>
<html>
<body>
<script>
  const remote = require('electron').remote;
  console.log(remote.app.getVersion());
</script>
</body>
</html>
```

如果想运行应用，参考 `Run your app` 。

解构任务

如果你使用的是CoffeeScript或Babel，你可以使用destructuring assignment来让使用内置模块更简单:

```
const {app, BrowserWindow} = require('electron');
```

然而如果你使用的是普通的JavaScript，你就需要等到Chrome支持ES6了。

使用内置模块时禁用旧样式

在版本v0.35.0之前，所有的内置模块都需要按造 `require('module-name')` 形式来使用，虽然它有很多弊端，我们仍然在老的应用中友好的支持它。

为了完整的禁用旧样式，你可以设置环境变量 `ELECTRON_HIDE_INTERNAL_MODULES` ：

```
process.env.ELECTRON_HIDE_INTERNAL_MODULES = 'true'
```

或者调用 `hideInternalModules` API:

```
require('electron').hideInternalModules()
```

进程

Electron 中的 `process` 对象 与 upstream node 中的有以下的不同点:

- `process.type` String - 进程类型, 可以是 `browser` (i.e. main process) 或 `renderer` .
- `process.versions['electron']` String - Electron的版本.
- `process.versions['chrome']` String - Chromium的版本.
- `process.resourcesPath` String - JavaScript源代码路径.
- `process.mas` Boolean - 在Mac App Store 创建, 它的值为 `true` , 在其它的地方值为 `undefined` .

事件

事件: 'loaded'

在Electron已经加载了其内部预置脚本和它准备加载主进程或渲染进程的时候触发.

当node被完全关闭的时候，它可以被预加载脚本使用来添加(原文: removed)与node无关的全局符号来回退到全局范围:

```
// preload.js
var _setImmediate = setImmediate;
var _clearImmediate = clearImmediate;
process.once('loaded', function() {
  global.setImmediate = _setImmediate;
  global.clearImmediate = _clearImmediate;
});
```

属性

`process.noAsar`

设置它为 `true` 可以使 `asar` 文件在node的内置模块中实效.

方法

`process` 对象有如下方法:

`process.hang()`

使当前进程的主线成挂起.

`process.setFdLimit(maxDescriptors)` *OS X Linux*

- `maxDescriptors` Integer

设置文件描述符软限制于 `maxDescriptors` 或硬限制与os, 无论它是否低于当前进程.

支持的 Chrome 命令行开关

这页列出了Chrome浏览器和Electron支持的命令行开关. 你也可以在app模块的ready事件发出之前使用app.commandLine.appendSwitch 来添加它们到你应用的main脚本里面:

```
const app = require('electron').app;
app.commandLine.appendSwitch('remote-debugging-port', '8315');
app.commandLine.appendSwitch('host-rules', 'MAP * 127.0.0.1');

app.on('ready', function() {
  // Your code here
});
```

--client-certificate= `path`

设置客户端的证书文件 `path` .

--ignore-connections-limit= `domains`

忽略用 `,` 分隔的 `domains` 列表的连接限制.

--disable-http-cache

禁止请求 HTTP 时使用磁盘缓存.

--remote-debugging-port= `port`

在指定的 `端口` 通过 HTTP 开启远程调试.

--js-flags= `flags`

指定引擎过渡到 JS 引擎.

在启动Electron时，如果你想在主进程中激活 `flags` ，它将被转换.

```
$ electron --js-flags="--harmony_proxies --harmony_collections" your-app
```

--proxy-server= `address:port`

使用一个特定的代理服务器，它将比系统设置的优先级更高.这个开关只有在使用 HTTP 协议时有效，它包含 HTTPS 和 WebSocket 请求. 值得注意的是，不是所有的代理服务器都支持 HTTPS 和 WebSocket 请求.

--proxy-bypass-list= `hosts`

让 Electron 使用(原文:bypass) 提供的以 semi-colon 分隔的hosts列表的代理服务器.这个开关只有在使用 `--proxy-server` 时有效.

例如:

```
app.commandLine.appendSwitch('proxy-bypass-list', '<local>;*.google.com;*foo.com;1.2.3.4:
```

将会为所有的hosts使用代理服务器，除了本地地址 (`localhost` , `127.0.0.1` etc.), `google.com` 子域, 以 `foo.com` 结尾的hosts，和所有类似 `1.2.3.4:5678` 的.

--proxy-pac-url= `url`

在指定的 `url` 上使用 PAC 脚本.

--no-proxy-server

不使用代理服务并且总是使用直接连接.忽略所有的合理代理标志.

--host-rules= `rules`

一个逗号分隔的 `rule` 列表来控制主机名如何映射.

例如:

- `MAP * 127.0.0.1` 强制所有主机名映射到 127.0.0.1
- `MAP *.google.com proxy` 强制所有 google.com 子域 使用 "proxy".
- `MAP test.com [::1]:77` 强制 "test.com" 使用 IPv6 回环地址. 也强制使用端口 77.
- `MAP * baz, EXCLUDE www.google.com` 重新全部映射到 "baz", 除了 "www.google.com".

这些映射适用于终端网络请求 (TCP 连接 和 主机解析 以直接连接的方式, 和 `CONNECT` 以代理连接, 还有 终端 host 使用 `SOCKS` 代理连接).

--host-resolver-rules= `rules`

类似 `--host-rules` ，但是 `rules` 只适合主机解析.

--ignore-certificate-errors

忽略与证书相关的错误.

--ppapi-flash-path= `path`

设置Pepper Flash插件的路径 `path` .

--ppapi-flash-version= `version`

设置Pepper Flash插件版本号.

--log-net-log= `path`

使网络日志事件能够被读写到 `path` .

--ssl-version-fallback-min= `version`

设置最简化的 SSL/TLS 版本号 ("tls1", "tls1.1" or "tls1.2")，TLS 可接受回退.

--cipher-suite-blacklist= `cipher_suites`

指定逗号分隔的 SSL 密码套件 列表实效.

--disable-renderer-backgrounding

防止 Chromium 降低隐藏的渲染进程优先级.

这个标志对所有渲染进程全局有效，如果你只想在一个窗口中禁止使用，你可以采用 hack 方法playing silent audio.

--enable-logging

打印 Chromium 信息输出到控制台.

如果在用户应用加载完成之前解析 `app.commandLine.appendSwitch` ，这个开关将实效，但是你可以设置 `ELECTRON_ENABLE_LOGGING` 环境变量来达到相同的效果.

--v= `log_level`

设置默认最大活跃 V-logging 标准; 默认为 0.通常 V-logging 标准值为肯定值.

这个开关只有在 `--enable-logging` 开启时有效.

--vmodule= `pattern`

赋予每个模块最大的 V-logging levels 来覆盖 `--v` 给的值.E.g. `my_module=2,foo*=3` 会改变所有源文件 `my_module.*` and `foo*.*` 的代码中的 logging level .

任何包含向前的(forward slash)或者向后的(backward slash)模式将被测试用于阻止整个路径名，并且不仅是E.g模块. `*/foo/bar/*=2` 将会改变所有在 `foo/bar` 下的源文件代码中的 logging level .

这个开关只有在 `--enable-logging` 开启时有效.

环境变量

一些 Electron 的行为受到环境变量的控制，因为他们的初始化比命令行和应用代码更早.

POSIX shells 的例子:

```
$ export ELECTRON_ENABLE_LOGGING=true
$ electron
```

Windows 控制台:

```
> set ELECTRON_ENABLE_LOGGING=true
> electron
```

`ELECTRON_RUN_AS_NODE`

类似node.js普通进程启动方式.

`ELECTRON_ENABLE_LOGGING`

打印 Chrome 的内部日志到控制台.

`ELECTRON_LOG_ASAR_READS`

当 Electron 读取 ASA 文档，把 read offset 和文档路径做日志记录到系统 `tmpdir`.结果文件将提供给 ASAR 模块来优化文档组织.

`ELECTRON_ENABLE_STACK_DUMPING`

当 Electron 崩溃的时候，打印堆栈记录到控制台.

如果 `crashReporter` 已经启动那么这个环境变量实效.

`ELECTRON_DEFAULT_ERROR_MODE` *Windows*

当 Electron 崩溃的时候，显示windows的崩溃对话框.

如果 `crashReporter` 已经启动那么这个环境变量实效.

`ELECTRON_NO_ATTACH_CONSOLE` *Windows*

不可使用当前控制台.

`ELECTRON_FORCE_WINDOW_MENU_BAR` *Linux*

不可再 Linux 上使用全局菜单栏.

`ELECTRON_HIDE_INTERNAL_MODULES`

关闭旧的内置模块如 `require('ipc')` 的通用模块.

自定义的 DOM 元素

`File` 对象

为了让用户能够通过HTML5的file API直接操作本地文件，DOM的File接口提供了对本地文件的抽象。Electron在File接口中增加了一个path属性，它是文件在系统中的真实路径。

获取拖动到APP中文件的真实路径的例子：

```
<div id="holder">
  Drag your file here
</div>

<script>
  var holder = document.getElementById('holder');
  holder.ondragover = function () {
    return false;
  };
  holder.ondragleave = holder.ondragend = function () {
    return false;
  };
  holder.ondrop = function (e) {
    e.preventDefault();
    var file = e.dataTransfer.files[0];
    console.log('File you dragged here is', file.path);
    return false;
  };
</script>
```

`<webview>` 标签

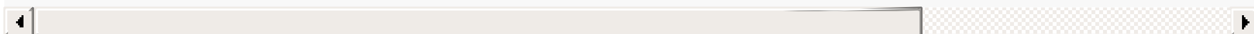
使用 `webview` 标签来把 'guest' 内容（例如 web pages ）嵌入到你的 Electron app 中. guest 内容包含在 `webview` 容器中.一个嵌入你应用的page控制着guest内容如何布局摆放和表达含义.

与 `iframe` 不同, `webview` 和你的应用运行的是不同的进程. 它不拥有渲染进程的权限，并且应用和嵌入内容之间的交互全部都是异步的.因为这能保证应用的安全性不受嵌入内容的影响.

例子

把一个 web page 嵌入到你的app，首先添加 `webview` 标签到你的app待嵌入page(展示 guest content). 在一个最简单的 `webview` 中，它包含了 web page 的文件路径和一个控制 `webview` 容器展示效果的css样式:

```
<webview id="foo" src="https://www.github.com/" style="display:inline-block; width:640px;
```



如果想随时控制 guest 内容，可以添加 JavaScript 脚本来监听 `webview` 事件使用 `webview` 方法来做出响应. 这里是2个事件监听的例子：一个监听 web page 准备加载，另一个监听 web page 停止加载，并且在加载的时候显示一条 "loading..." 信息:

```
<script>
  onload = function() {
    var webview = document.getElementById("foo");
    var indicator = document.querySelector(".indicator");

    var loadstart = function() {
      indicator.innerText = "loading...";
    }
    var loadstop = function() {
      indicator.innerText = "";
    }
    webview.addEventListener("did-start-loading", loadstart);
    webview.addEventListener("did-stop-loading", loadstop);
  }
</script>
```

标签属性

`webview` 标签有下面一些属性：

`src`

```
<webview src="https://www.github.com/"></webview>
```

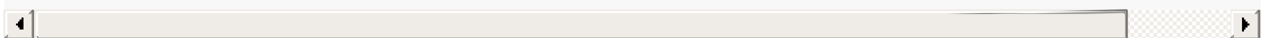
将一个可用的url做为这个属性的初始值添加到顶部导航.

如果把当前页的src添加进去将加载当前page.

`src` 同样可以填 data urls，例如 `data:text/plain,Hello, world!` .

`autosize`

```
<webview src="https://www.github.com/" autosize="on" minwidth="576" minheight="432"></web
```



如果这个属性的值为 "on" ， `webview` 容器将会根据属性 `minwidth` , `minheight` , `maxwidth` , 和 `maxheight` 的值在它们之间自适应. 只有在 `autosize` 开启的时候这个约束才会有效. 当 `autosize` 开启的时候， `webview` 容器的 size 只能在上面那四个属性值之间.

`nodeintegration`

```
<webview src="http://www.google.com/" nodeintegration></webview>
```

如果设置了这个属性， `webview` 中的 guest page 将整合node，并且拥有可以使用系统底层的资源，例如 `require` 和 `process` .

`plugins`

```
<webview src="https://www.github.com/" plugins></webview>
```

如果这个属性的值为 "on" ， `webview` 中的 guest page 就可以使用浏览器插件。

`preload`

```
<webview src="https://www.github.com/" preload="./test.js"></webview>
```

在 guest page 中的其他脚本执行之前预加载一个指定的脚本。规定预加载脚本的url须如 `file:` 或者 `asar:` ，因为它在是 guest page 中通过通过 `require` 命令加载的。

如果 guest page 没有整合 node ，这个脚本将试图使用真个 Node APIs ，但是在这个脚本执行完毕时，之前由node插入的全局对象会被删除。

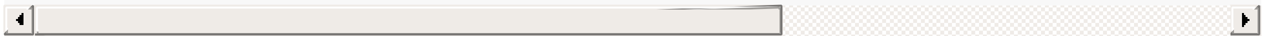
`httpreferrer`

```
<webview src="https://www.github.com/" httpreferrer="http://cheng.guru"></webview>
```

为 guest page 设置 referrer URL。

`useragent`

```
<webview src="https://www.github.com/" useragent="Mozilla/5.0 (Windows NT 6.1; WOW64; Tri
```



在 guest page 加载之前为其设置用户代理。如果页面已经加载了，可以使用 `setUserAgent` 方法来改变用户代理。

`disablewebsecurity`

```
<webview src="https://www.github.com/" disablewebsecurity></webview>
```

如果这个属性的值为 "on" ， guest page会禁用web安全控制.

partition

```
<webview src="https://github.com" partition="persist:github"></webview>
<webview src="http://electron.atom.io" partition="electron"></webview>
```

为page设置session。如果初始值为 `partition` ,这个 page 将会为app中的所有 page 应用同一个持续有效的 session。如果没有 `persist:` 前缀, 这个 page 将会使用一个历史 session 。通过分配使用相同的 `partition` , 所有的page都可以分享相同的session。如果 `partition` 没有设置，那app将使用默认的session.

这个值只能在在第一个渲染进程之前设置修改，之后修改的话会无效并且抛出一个DOM异常.

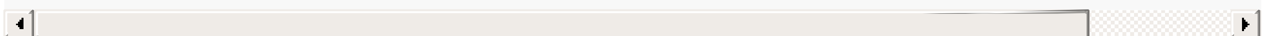
`allowpopups`

```
<webview src="https://www.github.com/" allowpopups></webview>
```

如果这个属性的值为 "on" ，将允许 guest page 打开一个新窗口。

`blinkfeatures`

```
<webview src="https://www.github.com/" blinkfeatures="PreciseMemoryInfo, CSSVariables"></
```



这个属性的值为一个用逗号分隔的列表，它的值指定特性被启用。你可以从 setFeatureEnabledFromString函数找到完整的支持特性。

方法

`webview` 的方法集合:

注意: webview 元素必须在使用这些方法之前加载完毕。

例如

```
webview.addEventListener("dom-ready", function() {
  webview.openDevTools();
});
```

`<webview>.loadURL(url[, options])`

- `url` URL
- `options` Object (可选)
  - `httpReferrer` String - 一个http类型的url.
  - `userAgent` String -用于发起请求的用户代理.
  - `extraHeaders` String - 额外的headers,用 "\n"分隔.

加载 webview 中的 `url` ，`url` 必须包含协议前缀，例如 `http://` 或 `file://`.

`<webview>.getURL()`

从 guest page 中返回 url.

`<webview>.getTitle()`

从 guest page 中返回 title.

`<webview>.isLoading()`

返回一个 guest page 是否仍在加载资源的布尔值.

`<webview>.isWaitingForResponse()`

返回一个 guest page 是否正在等待page的主要资源做出回应的布尔值.

`<webview>.stop()`

停止渲染.

`<webview>.reload()`

重新加载 guest page.

`<webview>.reloadIgnoringCache()`

忽视缓存，重新加载 guest page.

`<webview>.canGoBack()`

返回一个 guest page 是否能够回退的布尔值.

`<webview>.canGoForward()`

返回一个 guest page 是否能够前进的布尔值.

`<webview>.canGoToOffset(offset)`

- `offset` Integer

返回一个 guest page 是否能够前进到 `offset` 的布尔值.

`<webview>.clearHistory()`

清除导航历史.

`<webview>.goBack()`

guest page 回退.

`<webview>.goForward()`

guest page 前进.

`<webview>.goToIndex(index)`

- `index` Integer

guest page 导航到指定的绝对位置.

`<webview>.goToOffset(offset)`

- `offset` Integer

guest page 导航到指定的相对位置.

`<webview>.isCrashed()`

返回一个 渲染进程是否崩溃 的布尔值.

`<webview>.setUserAgent(userAgent)`

- `userAgent` String

重新设置用户代理.

`<webview>.getUserAgent()`

返回用户代理名字，返回类型： `String` .

`<webview>.insertCSS(css)`

- `css` String

插入css.

`<webview>.executeJavaScript(code, userGesture, callback)`

- `code` String
- `userGesture` Boolean - 默认 `false` .
- `callback` Function (可选) - 回调函数.
    - `result`

评估 `code` ，如果 `userGesture` 值为 true ，它将在这个page里面创建用户手势. HTML APIs ，如 `requestFullScreen` ,它需要用户响应，那么将自动通过这个参数优化.

`<webview>.openDevTools()`

为 guest page 打开开发工具调试窗口.

`<webview>.closeDevTools()`

为 guest page 关闭开发工具调试窗口.

`<webview>.isDevToolsOpened()`

返回一个 guest page 是否打开了开发工具调试窗口的布尔值.

`<webview>.isDevToolsFocused()`

返回一个 guest page 是否聚焦了开发工具调试窗口的布尔值.

`<webview>.inspectElement(x, y)`

- `x` Integer
- `y` Integer

开始检查 guest page 在 (`x` , `y`) 位置的元素.

`<webview>.inspectServiceWorker()`

在 guest page 中为服务人员打开开发工具.

`<webview>.setAudioMuted(muted)`

- `muted` Boolean 设置 guest page 流畅(muted).

`<webview>.isAudioMuted()`

返回一个 guest page 是否流畅的布尔值.

`<webview>.undo()`

在page中编辑执行 `undo` 命令.

`<webview>.redo()`

在page中编辑执行 `redo` 命令.

`<webview>.cut()`

在page中编辑执行 `cut` 命令.

`<webview>.copy()`

在page中编辑执行 `copy` 命令.

`<webview>.paste()`

在page中编辑执行 `paste` 命令.

`<webview>.pasteAndMatchStyle()`

在page中编辑执行 `pasteAndMatchStyle` 命令.

`<webview>.delete()`

在page中编辑执行 `delete` 命令.

`<webview>.selectAll()`

在page中编辑执行 `selectAll` 命令.

`<webview>.unselect()`

在page中编辑执行 `unselect` 命令.

`<webview>.replace(text)`

- `text` String

在page中编辑执行 `replace` 命令.

`<webview>.replaceMisspelling(text)`

- `text` String

在page中编辑执行 `replaceMisspelling` 命令.

`<webview>.insertText(text)`

- `text` String

插入文本.

`<webview>.findInPage(text[, options])`

- `text` String - 搜索内容,不能为空.
- `options` Object (可选)
  - `forward` Boolean - 向前或向后, 默认为 `true` .
  - `findNext` Boolean - 是否查找的第一个结果, 默认为 `false` .
  - `matchCase` Boolean - 是否区分大小写, 默认为 `false` .
  - `wordStart` Boolean - 是否只查找首字母. 默认为 `false` .
  - `medialCapitalAsWordStart` Boolean - 当配合 `wordStart` 的时候,接受一个文字中的匹配项，要求匹配项是以大写字母开头后面跟小写字母或者没有字母。可以接受一些其他单词内部匹配, 默认为 `false` .

发起一个请求来寻找页面中的所有匹配 `text` 的地方并且返回一个 `Integer` 来表示这个请求用的请求Id. 这个请求结果可以通过订阅 `found-in-page` 事件来取得.

`<webview>.stopFindInPage(action)`

- `action` String - 指定一个行为来接替停止 `<webview>.findInPage` 请求.
  - `clearSelection` - 转变为一个普通的 selection.
  - `keepSelection` - 清除 selection.
  - `activateSelection` - 聚焦并点击 selection node.

使用 `action` 停止 `findInPage` 请求.

`<webview>.print([options])`

打印输出 `webview` 的 web page. 类似 `webContents.print([options])` .

`<webview>.printToPDF(options, callback)`

以pdf格式打印输出 `webview` 的 web page. 类似 `webContents.printToPDF(options, callback)` .

`<webview>.send(channel[, arg1][, arg2][, ...])`

- `channel` String
- `arg` (可选)

通过 `channel` 向渲染进程发出异步消息，你也可以发送任意的参数。 渲染进程通过 `ipcRenderer` 模块监听 `channel` 事件来控制消息.

例子 webContents.send.

`<webview>.sendInputEvent(event)`

- `event` Object

向 page 发送输入事件.

查看 webContents.sendInputEvent 关于 `event` 对象的相信介绍.

`<webview>.getWebContents()`

返回和这个 `webview` 相关的 WebContents.

DOM 事件

`webview` 可用下面的 DOM 事件:

Event: 'load-commit'

返回:

- `url` String
- `isMainFrame` Boolean

加载完成触发. 这个包含当前文档的导航和副框架的文档加载，但是不包含异步资源加载.

Event: 'did-finish-load'

在导航加载完成时触发，也就是tab 的 spinner停止spinning，并且加载事件处理.

Event: 'did-fail-load'

Returns:

- `errorCode` Integer
- `errorDescription` String
- `validatedURL` String

类似 `did-finish-load` ，在加载失败或取消是触发，例如提出 `window.stop()` .

Event: 'did-frame-finish-load'

返回:

- `isMainFrame` Boolean

当一个 frame 完成 加载时触发.

Event: 'did-start-loading'

开始加载时触发.

Event: 'did-stop-loading'

停止家在时触发.

Event: 'did-get-response-details'

返回:

- `status` Boolean
- `newURL` String
- `originalURL` String

- `httpResponseCode` Integer
- `requestMethod` String
- `referrer` String
- `headers` Object

当获得返回详情的时候触发.

`status` 指示 socket 连接来下载资源.

Event: 'did-get-redirect-request'

返回:

- `oldURL` String
- `newURL` String
- `isMainFrame` Boolean

当重定向请求资源被接收的时候触发.

Event: 'dom-ready'

当指定的frame文档加载完毕时触发.

Event: 'page-title-updated'

返回:

- `title` String
- `explicitSet` Boolean

当导航中的页面title被设置时触发. 在title通过文档路径异步加载时 `explicitSet` 为false.

Event: 'page-favicon-updated'

返回:

- `favicons` Array - Array of URLs.

当page收到了图标url时触发.

Event: 'enter-html-full-screen'

当通过HTML API使界面进入全屏时触发.

Event: 'leave-html-full-screen'

当通过HTML API使界面退出全屏时触发.

Event: 'console-message'

返回:

- `level` Integer
- `message` String
- `line` Integer
- `sourceId` String

当客户端输出控制台信息的时候触发.

下面示例代码将所有信息输出到内置控制台，没有考虑到输出等级和其他属性。

```
webview.addEventListener('console-message', function(e) {
  console.log('Guest page logged a message:', e.message);
});
```

Event: 'found-in-page'

返回:

- `result` Object
  - `requestId` Integer
  - `finalUpdate` Boolean - 指明下面是否还有更多的回应.
  - `activeMatchOrdinal` Integer (可选) - 活动匹配位置
  - `matches` Integer (optional) - 匹配数量.
  - `selectionArea` Object (optional) - 整合第一个匹配域.

在请求 `webview.findInPage` 结果有效时触发.

```
webview.addEventListener('found-in-page', function(e) {
  if (e.result.finalUpdate)
    webview.stopFindInPage("keepSelection");
});

const rqustId = webview.findInPage("test");
```

Event: 'new-window'

返回:

- `url` String

- `frameName` String
- `disposition` String - 可以为 `default` , `foreground-tab` , `background-tab` , `new-window` 和 `other` .
- `options` Object - 参数应该被用作创建新的 `BrowserWindow` .

在 guest page 试图打开一个新的浏览器窗口时触发.

下面示例代码在系统默认浏览器中打开了一个新的url.

```
webview.addEventListener('new-window', function(e) {
  require('electron').shell.openExternal(e.url);
});
```

Event: 'will-navigate'

返回:

- `url` String

当用户或page尝试开始导航时触发. 它能在 `window.location` 变化或者用户点击连接的时候触发.

这个事件在以 APIS 编程方式开始导航时不会触发，例如 `<webview>.loadURL` 和 `<webview>.back` .

在页面内部导航跳转也将不回触发这个事件，例如点击锚链接或更新 `window.location.hash` . 使用 `did-navigate-in-page` 来实现页内跳转事件.

使用 `event.preventDefault()` 并不会起什么用.

Event: 'did-navigate'

返回:

- `url` String

当导航结束时触发.

在页面内部导航跳转也将不回触发这个事件，例如点击锚链接或更新 `window.location.hash` . 使用 `did-navigate-in-page` 来实现页内跳转事件.

Event: 'did-navigate-in-page'

返回:

- `url` String

当页内导航发生时触发. 当业内导航发生时，page url改变了，但是不会跳出 page . 例如在锚链接被电击或DOM `hashchange` 事件发生时触发.

Event: 'close'

在 guest page试图关闭自己的时候触发.

下面的示例代码指示了在客户端试图关闭自己的时候将改变导航连接为 `about:blank` .

```
webview.addEventListener('close', function() {
  webview.src = 'about:blank';
});
```

Event: 'ipc-message'

返回:

- `channel` String
- `args` Array

在 guest page 向嵌入页发送一个异步消息的时候触发.

你可以很简单的使用 `sendToHost` 方法和 `ipc-message` 事件在 guest page 和 嵌入页(embedder page)之间通信:

```
// In embedder page.
webview.addEventListener('ipc-message', function(event) {
  console.log(event.channel);
  // Prints "pong"
});
webview.send('ping');
```

```
// In guest page.
var ipcRenderer = require('electron').ipcRenderer;
ipcRenderer.on('ping', function() {
  ipcRenderer.sendToHost('pong');
});
```

Event: 'crashed'

在渲染进程崩溃的时候触发.

Event: 'gpu-crashed'

在GPU进程崩溃的时候触发.

Event: 'plugin-crashed'

返回:

- `name` String
- `version` String

在插件进程崩溃的时候触发.

Event: 'destroyed'

在界面内容销毁的时候触发.

Event: 'media-started-playing'

在媒体准备播放的时候触发.

Event: 'media-paused'

在媒体暂停播放或播放放毕的时候触发.

Event: 'did-change-theme-color'

在页面的主体色改变的时候触发. 在使用 meta 标签的时候这就很常见了:

```
<meta name='theme-color' content='#ff0000'>
```

Event: 'devtools-opened'

在开发者工具打开的时候触发.

Event: 'devtools-closed'

在开发者工具关闭的时候触发.

Event: 'devtools-focused'

在开发者工具获取焦点的时候触发.

`window.open` 函数

当在界面中使用 `window.open` 来创建一个新的窗口时候，将会创建一个 `BrowserWindow` 的实例，并且将返回一个标识，这个界面通过标识来对这个新的窗口进行有限的控制.

这个标识对传统的web界面来说，通过它能对子窗口进行有限的功能性兼容控制. 想要完全的控制这个窗口，可以直接创建一个 `BrowserWindow` .

新创建的 `BrowserWindow` 默认为继承父窗口的属性参数，想重写属性的话可以在 `features` 中设置他们.

`window.open(url[, frameName][, features])`

- `url` String
- `frameName` String (可选)
- `features` String (可选)

创建一个新的window并且返回一个 `BrowserWindowProxy` 类的实例.

`features` 遵循标准浏览器的格式，但是每个feature 应该作为 `BrowserWindow` 参数的一个字段.

`window.opener.postMessage(message, targetOrigin)`

- `message` String
- `targetOrigin` String

通过指定位置或用 `*` 来代替没有明确位置来向父窗口发送信息.

Class: BrowserWindowProxy

`BrowserWindowProxy` 由 `window.open` 创建返回，并且提供了对子窗口的有限功能性控制.

`BrowserWindowProxy.blur()`

子窗口的失去焦点.

`BrowserWindowProxy.close()`

强行关闭子窗口，忽略卸载事件.

`BrowserWindowProxy.closed`

在子窗口关闭之后恢复正常.

`BrowserWindowProxy.eval(code)`

- `code` String

评估子窗口的代码.

`BrowserWindowProxy.focus()`

子窗口获得焦点(让其显示在最前).

`BrowserWindowProxy.postMessage(message, targetOrigin)`

- `message` String
- `targetOrigin` String

通过指定位置或用 `*` 来代替没有明确位置来向子窗口发送信息.

除了这些方法，子窗口还可以无特性和使用单一方法来实现 `window.opener` 对象.

在主进程内可用的模块

app

`app` 模块是为了控制整个应用的生命周期设计的。

下面的这个例子将会展示如何在最后一个窗口被关闭时退出应用：

```
var app = require('app');
app.on('window-all-closed', function() {
  app.quit();
});
```

事件列表

`app` 对象会触发以下的事件：

事件：'will-finish-launching'

当应用程序完成基础的启动的时候被触发。在 Windows 和 Linux 中，`will-finish-launching` 事件与 `ready` 事件是相同的；在 OS X 中，这个时间相当于 `NSApplication` 中的 `applicationWillFinishLaunching` 提示。你应该经常在这里为 `open-file` 和 `open-url` 设置监听器，并启动崩溃报告和自动更新。

在大多数的情况下，你应该只在 `ready` 事件处理器中完成所有的业务。

事件：'ready'

当 Electron 完成初始化时被触发。

事件：'window-all-closed'

当所有的窗口都被关闭时触发。

这个时间仅在应用还没有退出时才能触发。如果用户按下了 `Cmd + Q`，或者开发者调用了 `app.quit()`，Electron 将会先尝试关闭所有的窗口再触发 `will-quit` 事件，在这种情况下 `window-all-closed` 不会被触发。

事件：'before-quit'

返回：

- `event` Event

在应用程序开始关闭它的窗口的时候被触发。 调用 `event.preventDefault()` 将会阻止终止应用程序的默认行为。

事件：'will-quit'

返回：

- `event` Event

当所有的窗口已经被关闭，应用即将退出时被触发。 调用 `event.preventDefault()` 将会阻止终止应用程序的默认行为。

你可以在 `window-all-closed` 事件的描述中看到 `will-quit` 事件 和 `window-all-closed` 事件的区别。

事件：'quit'

返回：

- `event` Event
- `exitCode` Integer

当应用程序正在退出时触发。

事件：'open-file' *OS X*

返回：

- `event` Event
- `path` String

当用户想要在应用中打开一个文件时触发。`open-file` 事件常常在应用已经打开并且系统想要再次使用应用打开文件时被触发。 `open-file` 也会在一个文件被拖入 dock 且应用还没有运行的时候被触发。 请确认在应用启动的时候（甚至在 `ready` 事件被触发前）就对 `open-file` 事件进行监听，以处理这种情况。

如果你想处理这个事件，你应该调用 `event.preventDefault()` 。 在 Windows系统中, 你需要通过解析 process.argv 来获取文件路径。

事件：'open-url' *OS X*

返回：

- `event` Event
- `url` String

当用户想要在应用中打开一个url的时候被触发。URL格式必须要提前标识才能被你的应用打开。

如果你想处理这个事件，你应该调用 `event.preventDefault()` 。

事件：'activate' *OS X*

返回：

- `event` Event
- `hasVisibleWindows` Boolean

当应用被激活时触发，常用于点击应用的 dock 图标的时候。

事件：'browser-window-blur'

返回：

- `event` Event
- `window` BrowserWindow

当一个 BrowserWindow 失去焦点的时候触发。

事件：'browser-window-focus'

返回：

- `event` Event
- `window` BrowserWindow

当一个 BrowserWindow 获得焦点的时候触发。

事件：'browser-window-created'

返回：

- `event` Event
- `window` BrowserWindow

当一个 BrowserWindow 被创建的时候触发。

事件：'certificate-error'

返回：

- `event` Event

- `webContents` WebContents
- `url` String - URL 地址
- `error` String - 错误码
- `certificate` Object
  - `data` Buffer - PEM 编码数据
  - `issuerName` String - 发行者的公有名称
- `callback` Function

当对 `url` 验证 `certificate` 证书失败的时候触发，如果需要信任这个证书，你需要阻止默认行为 `event.preventDefault()` 并且 调用 `callback(true)` 。

```
session.on('certificate-error', function(event, webContents, url, error, certificate, cal
  if (url == "https://github.com") {
    // 验证逻辑。
    event.preventDefault();
    callback(true);
  } else {
    callback(false);
  }
});
```

事件：'select-client-certificate'

返回：

- `event` Event
- `webContents` WebContents
- `url` String - URL 地址
- `certificateList` [Object]
  - `data` Buffer - PEM 编码数据
  - `issuerName` String - 发行者的公有名称
- `callback` Function

当一个客户端认证被请求的时候被触发。

`url` 指的是请求客户端认证的网页地址，调用 `callback` 时需要传入一个证书列表中的证书。

需要通过调用 `event.preventDefault()` 来防止应用自动使用第一个证书进行验证。如下所示：

```
app.on('select-certificate', function(event, host, url, list, callback) {
  event.preventDefault();
  callback(list[0]);
})
```

事件: 'login'

返回：

- `event` Event
- `webContents` WebContents
- `request` Object
  - `method` String
  - `url` URL
  - `referrer` URL
- `authInfo` Object
  - `isProxy` Boolean
  - `scheme` String
  - `host` String
  - `port` Integer
  - `realm` String
- `callback` Function

当 `webContents` 要做进行一次 HTTP 登陆验证时被触发。

默认情况下，Electron 会取消所有的验证行为，如果需要重写这个行为，你需要用 `event.preventDefault()` 来阻止默认行为，并且 用 `callback(username, password)` 来进行验证。

```
app.on('login', function(event, webContents, request, authInfo, callback) {
  event.preventDefault();
  callback('username', 'secret');
})
```

事件：'gpu-process-crashed'

当 GPU 进程崩溃时触发。

方法列表

`app` 对象拥有以下的方法：

请注意 有的方法只能用于特定的操作系统。

`app.quit()`

试图关掉所有的窗口。 `before-quit` 事件将会最先被触发。如果所有的窗口都被成功关闭了， `will-quit` 事件将会被触发，默认下应用将会被关闭。

这个方法保证了所有的 `beforeunload` 和 `unload` 事件处理器被正确执行。假如一个窗口的 `beforeunload` 事件处理器返回 `false`，那么整个应用可能会取消退出。

`app.hide()` *OS X*

隐藏所有的应用窗口，不是最小化.

`app.show()` *OS X*

隐藏后重新显示所有的窗口，不会自动选中他们。

`app.exit(exitCode)`

- `exitCode` 整数

带着 `exitCode`退出应用.

所有的窗口会被立刻关闭，不会询问用户。 `before-quit` 和 `will-quit` 这2个事件不会被触发

`app.getAppPath()`

返回当前应用所在的文件路径。

`app.getPath(name)`

- `name` String

返回一个与 `name` 参数相关的特殊文件夹或文件路径。当失败时抛出一个 `Error` 。

你可以通过名称请求以下的路径：

- `home` 用户的 home 文件夹（主目录）
- `appData` 当前用户的应用数据文件夹，默认对应：
  - `%APPDATA%` Windows 中
  - `$XDG_CONFIG_HOME` or `~/.config` Linux 中
  - `~/Library/Application Support` OS X 中
- `userData` 储存你应用程序设置文件的文件夹，默认是 `appData` 文件夹附加应用的名称
- `temp` 临时文件夹
- `exe` 当前的可执行文件
- `module` `libchromiumcontent` 库
- `desktop` 当前用户的桌面文件夹
- `documents` 用户文档目录的路径
- `downloads` 用户下载目录的路径.

- `music` 用户音乐目录的路径.
- `pictures` 用户图片目录的路径.
- `videos` 用户视频目录的路径.

`app.setPath(name, path)`

- `name` String
- `path` String

重写某个 `name` 的路径为 `path`，`path` 可以是一个文件夹或者一个文件，这个和 `name` 的类型有关。 如果这个路径指向的文件夹不存在，这个文件夹将会被这个方法创建。 如果错误则会抛出 `Error`。

`name` 参数只能使用 `app.getPath` 中定义过 `name`。

默认情况下，网页的 cookie 和缓存都会储存在 `userData` 文件夹。 如果你想要改变这个位置，你需要在 `app` 模块中的 `ready` 事件被触发之前重写 `userData` 的路径。

`app.getVersion()`

返回加载应用程序的版本。如果应用程序的 `package.json` 文件中没有写版本号， 将会返回当前包或者可执行文件的版本。

`app.getName()`

返回当前应用程序的 `package.json` 文件中的名称。

由于 npm 的命名规则，通常 `name` 字段是一个短的小写字符串。但是应用名的完整名称通常是首字母大写的，你应该单独使用一个 `productName` 字段，用于表示你的应用程序的完整名称。Electron 会优先使用这个字段作为应用名。

`app.setName(name)`

- `name` String

重写当前应用的名字

`app.getLocale()`

返回当前应用程序的语言。

`app.addRecentDocument(path)` *OS X Windows*

- `path` String

在最近访问的文档列表中添加 `path` 。

这个列表由操作系统进行管理。在 Windows 中您可以通过任务条进行访问，在 OS X 中你可以通过 dock 菜单进行访问。

`app.clearRecentDocuments()` *OS X Windows*

清除最近访问的文档列表。

`app.setUserTasks(tasks)` *Windows*

- `tasks` [Task] - 一个由 Task 对象构成的数组

将 `tasks` 添加到 Windows 中 JumpList 功能的 Tasks 分类中。

`tasks` 中的 `Task` 对象格式如下：

`Task` Object

- `program` String - 执行程序的路径，通常你应该说明当前程序的路径为 `process.execPath` 字段。
- `arguments` String - 当 `program` 执行时的命令行参数。
- `title` String - JumpList 中显示的标题。
- `description` String - 对这个任务的描述。
- `iconPath` String - JumpList 中显示的图标的绝对路径，可以是一个任意包含一个图标的资源文件。通常来说，你可以通过指明 `process.execPath` 来显示程序中的图标。
- `iconIndex` Integer - 图标文件中的采用的图标位置。如果一个图标文件包括了多个图标，就需要设置这个值以确定使用的是哪一个图标。 如果这个图标文件中只包含一个图标，那么这个值为 0。

`app.allowNTLMCredentialsForAllDomains(allow)`

- `allow` Boolean

动态设置是否总是为 HTTP NTLM 或 Negotiate 认证发送证书。通常来说，Electron 只会对本地网络（比如和你处在一个域中的计算机）发 送 NTLM / Kerberos 证书。但是假如网络设置得不太好，可能这个自动探测会失效，所以你可以通过这个接口自定义 Electron 对所有 URL 的行为。

`app.makeSingleInstance(callback)`

- `callback` Function

这个方法可以让你的应用在同一时刻最多只会有一个实例，否则你的应用可以被运行多次并产生多个实例。你可以利用这个接口保证只有一个实例正 常运行，其余的实例全部会被终止并退出。

如果多个实例同时运行，那么第一个被运行的实例中 `callback` 会以 `callback(argv, workingDirectory)` 的形式被调用。其余的实例 会被终止。 `argv` 是一个包含了这个实例的命令行参数列表的数组， `workingDirectory` 是这个实例目前的运行目录。通常来说，我们会用通过将应用在 主屏幕上激活，并且取消最小化，来提醒用户这个应用已经被打开了。

在 `app` 的 `ready` 事件后， `callback` 才有可能被调用。

如果当前实例为第一个实例，那么在这个方法将会返回 `false` 来保证它继续运行。否则将会返回 `true` 来让它立刻退出。

在 OS X 中，如果用户通过 Finder、 `open-file` 或者 `open-url` 打开应用，系统会强制确保只有一个实例在运行。但是如果用户是通过 命令行打开，这个系统机制会被忽略，所以你仍然需要靠这个方法来保证应用为单实例运行的。

下面是一个简单的例子。我们可以通过这个例子了解如何确保应用为单实例运行状态。

```
var myWindow = null;

var shouldQuit = app.makeSingleInstance(function(commandLine, workingDirectory) {
  // 当另一个实例运行的时候，这里将会被调用，我们需要激活应用的窗口
  if (myWindow) {
    if (myWindow.isMinimized()) myWindow.restore();
    myWindow.focus();
  }
  return true;
});

// 这个实例是多余的实例，需要退出
if (shouldQuit) {
  app.quit();
  return;
}

// 创建窗口、继续加载应用、应用逻辑等……
app.on('ready', function() {
});
```

`app.setAppUserModelId(id)` *Windows*

- `id` String

改变当前应用的 Application User Model ID 为 `id` .

`app.isAeroGlassEnabled()` *Windows*

如果 DWM composition(Aero Glass) 启用 了，那么这个方法会返回 `true`，否则是 `false`。你可以用这个方法来决定是否要开启透明窗口特效，因为如果用户没开启 DWM，那么透明窗 口特效是无效的。

举个例子：

```
let browserOptions = {width: 1000, height: 800};

// 只有平台支持的时候才使用透明窗口
if (process.platform !== 'win32' || app.isAeroGlassEnabled()) {
  browserOptions.transparent = true;
  browserOptions.frame = false;
}

// 创建窗口
win = new BrowserWindow(browserOptions);

// 转到某个网页
if (browserOptions.transparent) {
  win.loadURL('file://' + __dirname + '/index.html');
} else {
  // 没有透明特效，我们应该用某个只包含基本样式的替代解决方案。
  win.loadURL('file://' + __dirname + '/fallback.html');
}
```

`app.commandLine.appendSwitch(switch[, value])`

通过可选的参数 `value` 给 Chromium 中添加一个命令行开关。

注意 这个方法不会影响 `process.argv`，我们通常用这个方法控制一些底层 Chromium 行为。

`app.commandLine.appendArgument(value)`

给 Chromium 中直接添加一个命令行参数，这个参数 `value` 的引号和格式必须正确。

注意 这个方法不会影响 `process.argv`。

`app.dock.bounce([type])` *OS X*

- `type` String - 可选参数，可以是 `critical` 或 `informational`。默认为 `informational`。

当传入的是 `critical` 时，dock 中的应用将会开始弹跳，直到这个应用被激活或者这个请求被取消。

当传入的是 `informational` 时，dock 中的图标只会弹跳一秒钟。但是，这个请求仍然会激活，直到应用被激活或者请求被取消。

这个方法返回的返回值表示这个请求的 ID。

`app.dock.cancelBounce(id)` *OS X*

- `id` Integer

取消这个 `id` 对应的请求。

`app.dock.setBadge(text)` *OS X*

- `text` String

设置应用在 dock 中显示的字符串。

`app.dock.getBadge()` *OS X*

返回应用在 dock 中显示的字符串。

`app.dock.hide()` *OS X*

隐藏应用在 dock 中的图标。

`app.dock.show()` *OS X*

显示应用在 dock 中的图标。

`app.dock.setMenu(menu)` *OS X*

- `menu` Menu

设置应用的 dock 菜单.

`app.dock.setIcon(image)` *OS X*

- `image` NativeImage

设置应用在 dock 中显示的图标。

autoUpdater

这个模块提供了一个到 `Squirrel` 自动更新框架的接口。

平台相关的提示

虽然 `autoUpdater` 模块提供了一套各平台通用的接口，但是在每个平台间依然会有一些微小的差异。

OS X

在 OS X 上，`autoUpdater` 模块依靠的是内置的 Squirrel.Mac，这意味着你不需要依靠其他的设置就能使用。关于 更新服务器的配置，你可以通过阅读 Server Support 这篇文章来了解。

Windows

在 Windows 上，你必须使用安装程序将你的应用装到用户的计算机上，所以比较推荐的方法是用 grunt-electron-installer 这个模块来自动生成一个 Windows 安装向导。

Squirrel 自动生成的安装向导会生成一个带 Application User Model ID 的快捷方式。Application User Model ID 的格式是 `com.squirrel.PACKAGE_ID.YOUR_EXE_WITHOUT_DOT_EXE`，比如像 `com.squirrel.slack.Slack` 和 `com.squirrel.code.Code` 这样的。你应该在自己的应用中使用 `app.setAppUserModelId` 方法设置相同的 API，不然 Windows 将不能正确地把你的应用固定在任务栏上。

服务器端的配置和 OS X 也是不一样的，你可以阅读 Squirrel.Windows 这个文档来获得详细信息。

Linux

Linux 下没有任何的自动更新支持，所以我们推荐用各个 Linux 发行版的包管理器来分发你的应用。

事件列表

`autoUpdater` 对象会触发以下的事件：

事件：'error'

返回：

- `error` Error

当更新发生错误的时候触发。

事件：'checking-for-update'

当开始检查更新的时候触发。

事件：'update-available'

当发现一个可用更新的时候触发，更新包下载会自动开始。

事件：'update-not-available'

当没有可用更新的时候触发。

事件：'update-downloaded'

返回：

- `event` Event
- `releaseNotes` String - 新版本更新公告
- `releaseName` String - 新的版本号
- `releaseDate` Date - 新版本发布的日期
- `updateURL` String - 更新地址

在更新下载完成的时候触发。

在 Windows 上只有 `releaseName` 是有效的。

方法列表

`autoUpdater` 对象有以下的方法：

`autoUpdater.setFeedURL(url)`

- `url` String

设置检查更新的 `url`，并且初始化自动更新。这个 `url` 一旦设置就无法更改。

`autoUpdater.checkForUpdates()`

向服务端查询现在是否有可用的更新。在调用这个方法之前，必须要先调用 `setFeedURL`。

`autoUpdater.quitAndInstall()`

在下载完成后，重启当前的应用并且安装更新。这个方法应该仅在 `update-downloaded` 事件触发后被调用。

BrowserWindow

`BrowserWindow` 类让你有创建一个浏览器窗口的权力。例如:

```
// In the main process.
const BrowserWindow = require('electron').BrowserWindow;

// Or in the renderer process.
const BrowserWindow = require('electron').remote.BrowserWindow;

var win = new BrowserWindow({ width: 800, height: 600, show: false });
win.on('closed', function() {
  win = null;
});

win.loadURL('https://github.com');
win.show();
```

你也可以不通过chrome创建窗口，使用 Frameless Window API.

Class: BrowserWindow

`BrowserWindow` 是一个 EventEmitter.

通过 `options` 可以创建一个具有本质属性的 `BrowserWindow` .

`new BrowserWindow([options])`

- `options` Object
  - `width` Integer - 窗口宽度,单位像素. 默认是 `800` .
  - `height` Integer - 窗口高度,单位像素. 默认是 `600` .
  - `x` Integer - 窗口相对于屏幕的左偏移位置.默认居中.
  - `y` Integer - 窗口相对于屏幕的顶部偏移位置.默认居中.
  - `useContentSize` Boolean - `width` 和 `height` 使用web网页size, 这意味着实际窗口的size应该包括窗口框架的size，稍微会大一点，默认为 `false` .
  - `center` Boolean - 窗口屏幕居中.
  - `minWidth` Integer - 窗口最小宽度，默认为 `0` .
  - `minHeight` Integer - 窗口最小高度，默认为 `0` .
  - `maxWidth` Integer - 窗口最大宽度，默认无限制.
  - `maxHeight` Integer - 窗口最大高度，默认无限制.
  - `resizable` Boolean - 是否可以改变窗口size，默认为 `true` .
  - `movable` Boolean - 窗口是否可以拖动. 在 Linux 上无效. 默认为 `true` .
  - `minimizable` Boolean - 窗口是否可以最小化. 在 Linux 上无效. 默认为 `true` .
  - `maximizable` Boolean - 窗口是否可以最大化. 在 Linux 上无效. 默认为 `true` .

  - `closable` Boolean - 窗口是否可以关闭. 在 Linux 上无效. 默认为 `true` .
  - `alwaysOnTop` Boolean - 窗口是否总是显示在其他窗口之前. 在 Linux 上无效. 默认为 `false` .
  - `fullscreen` Boolean - 窗口是否可以全屏幕. 当明确设置值为When `false` ，全屏化按钮将会隐藏，在 OS X 将禁用. 默认 `false` .
  - `fullscreenable` Boolean - 在 OS X 上，全屏化按钮是否可用，默认为 `true` .
  - `skipTaskbar` Boolean - 是否在人物栏中显示窗口. 默认是 `false` .
  - `kiosk` Boolean - kiosk 方式. 默认为 `false` .
  - `title` String - 窗口默认title. 默认 `"Electron"` .
  - `icon` NativeImage - 窗口图标, 如果不设置，窗口将使用可用的默认图标.
  - `show` Boolean - 窗口创建的时候是否显示. 默认为 `true` .
  - `frame` Boolean - 指定 `false` 来创建一个 Frameless Window. 默认为 `true` .
  - `acceptFirstMouse` Boolean - 是否允许单击web view来激活窗口 . 默认为 `false` .
  - `disableAutoHideCursor` Boolean - 当 typing 时是否隐藏鼠标.默认 `false` .
  - `autoHideMenuBar` Boolean - 除非点击 `Alt` ，否则隐藏菜单栏.默认为 `false` .
  - `enableLargerThanScreen` Boolean - 是否允许允许改变窗口大小大于屏幕. 默认是 `false` .
  - `backgroundColor` String -窗口的 background color 值为十六进制,如 `#66CD00` 或 `#FFF` 或 `#80FFFFFF` (支持透明度). 默认为在 Linux 和 Windows 上为 `#000` (黑色) , Mac上为 `#FFF` (或透明).
  - `hasShadow` Boolean - 窗口是否有阴影. 只在 OS X 上有效. 默认为 `true` .
  - `darkTheme` Boolean - 为窗口使用 dark 主题, 只在一些拥有 GTK+3 桌面环境上有效. 默认为 `false` .
  - `transparent` Boolean - 窗口 透明. 默认为 `false` .
  - `type` String - 窗口type, 默认普通窗口. 下面查看更多.
  - `titleBarStyle` String - 窗口标题栏样式. 下面查看更多.
  - `webPreferences` Object - 设置界面特性. 下面查看更多.

`type` 的值和效果不同平台展示效果不同，具体:

- Linux, 可用值为 `desktop` , `dock` , `toolbar` , `splash` , `notification` .
- OS X, 可用值为 `desktop` , `textured` .
  - `textured` type 添加金属梯度效果 (`NSTexturedBackgroundWindowMask`).
  - `desktop` 设置窗口在桌面背景窗口水平 (`kCGDesktopWindowLevel - 1`). 注意桌面窗口不可聚焦, 不可不支持键盘和鼠标事件, 但是可以使用 `globalShortcut` 来解决输入问题.

`titleBarStyle` 只在 OS X 10.10 Yosemite 或更新版本上支持. 可用值:

- `default` 以及无值, 显示在 Mac 标题栏上为不透明的标准灰色.
- `hidden` 隐藏标题栏，内容充满整个窗口, 然后它依然在左上角，仍然受标准窗口控制.
- `hidden-inset` 主体隐藏，显示小的控制按钮在窗口边缘.

`webPreferences` 参数是个对象，它的属性:

- `nodeIntegration` Boolean - 是否完整支持node. 默认为 `true` .
- `preload` String - 界面的其它脚本运行之前预先加载一个指定脚本. 这个脚本将一直可以使用 node APIs 无论 node integration 是否开启. 脚本路径为绝对路径. 当 node integration 关闭, 预加载的脚本将从全局范围重新引入node的全局引用标志. 查看例子 here.
- `session` Session - 设置界面session. 而不是直接忽略session对象 , 也可用 `partition` 来代替, 它接受一个 partition 字符串. 当同时使用 `session` 和 `partition` , `session` 优先级更高. 默认使用默认 session.
- `partition` String - 通过session的partition字符串来设置界面session. 如果 `partition` 以 `persist:` 开头, 这个界面将会为所有界面使用相同的 `partition` . 如果没有 `persist:` 前缀, 界面使用历史session. 通过分享同一个 `partition` , 所有界面使用相同的session. 默认使用默认 session.
- `zoomFactor` Number - 界面默认缩放值, `3.0` 表示 `300%` . 默认 `1.0` .
- `javascript` Boolean - 开启javascript支持. 默认为 `true` .
- `webSecurity` Boolean - 当设置为 `false` , 它将禁用相同地方的规则 (通常测试服), 并且如果有2个非用户设置的参数，就设置 `allowDisplayingInsecureContent` 和 `allowRunningInsecureContent` 的值为 `true` . 默认为 `true` .
- `allowDisplayingInsecureContent` Boolean -允许一个使用 https的界面来展示由 http URLs 传过来的资源. 默认 `false` .
- `allowRunningInsecureContent` Boolean - Boolean -允许一个使用 https的界面来渲染由 http URLs 提交的html,css,javascript. 默认为 `false` .
- `images` Boolean - 开启图片使用支持. 默认 `true` .
- `textAreasAreResizable` Boolean - textArea 可以编辑. 默认为 `true` .
- `webgl` Boolean - 开启 WebGL 支持. 默认为 `true` .
- `webaudio` Boolean - 开启 WebAudio 支持. 默认为 `true` .
- `plugins` Boolean - 是否开启插件支持. 默认为 `false` .
- `experimentalFeatures` Boolean - 开启 Chromium 的 可测试 特性. 默认为 `false` .
- `experimentalCanvasFeatures` Boolean - 开启 Chromium 的 canvas 可测试特性. 默认为 `false` .
- `directWrite` Boolean - 开启窗口的 DirectWrite font 渲染系统. 默认为 `true` .
- `blinkFeatures` String - 以 `,` 分隔的特性列表, 如 `CSSVariables,KeyboardEventKey` . 被支持的所有特性可在 setFeatureEnabledFromString 中找到.
- `defaultFontFamily` Object - 设置 font-family 默认字体.
  - `standard` String - 默认为 `Times New Roman` .
  - `serif` String - 默认为 `Times New Roman` .
  - `sansSerif` String - 默认为 `Arial` .
  - `monospace` String - 默认为 `Courier New` .
- `defaultFontSize` Integer - 默认为 `16` .
- `defaultMonospaceFontSize` Integer - 默认为 `13` .

- `minimumFontSize` Integer - 默认为 `0` .
- `defaultEncoding` String - 默认为 `ISO-8859-1` .

事件

`BrowserWindow` 对象可触发下列事件:

注意: 一些事件只能在特定os环境中触发，已经尽可能地标出.

Event: 'page-title-updated'

返回:

- `event` Event

当文档改变标题时触发,使用 `event.preventDefault()` 可以阻止原窗口的标题改变.

Event: 'close'

返回:

- `event` Event

在窗口要关闭的时候触发. 它在DOM的 `beforeunload` and `unload` 事件之前触发.使用 `event.preventDefault()` 可以取消这个操作

通常你想通过 `beforeunload` 处理器来决定是否关闭窗口，但是它也会在窗口重载的时候被触发. 在 Electron 中，返回一个空的字符串或 `false` 可以取消关闭.例如:

```
window.onbeforeunload = function(e) {
  console.log('I do not want to be closed');

  // Unlike usual browsers, in which a string should be returned and the user is
  // prompted to confirm the page unload, Electron gives developers more options.
  // Returning empty string or false would prevent the unloading now.
  // You can also use the dialog API to let the user confirm closing the application.
  e.returnValue = false;
};
```

Event: 'closed'

当窗口已经关闭的时候触发.当你接收到这个事件的时候，你应当删除对已经关闭的窗口的引用对象和避免再次使用它.

Event: 'unresponsive'

在界面卡死的时候触发事件.

Event: 'responsive'

在界面恢复卡死的时候触发.

Event: 'blur'

在窗口失去焦点的时候触发.

Event: 'focus'

在窗口获得焦点的时候触发.

Event: 'maximize'

在窗口最大化的时候触发.

Event: 'unmaximize'

在窗口退出最大化的时候触发.

Event: 'minimize'

在窗口最小化的时候触发.

Event: 'restore'

在窗口从最小化恢复的时候触发.

Event: 'resize'

在窗口size改变的时候触发.

Event: 'move'

在窗口移动的时候触发.

注意：在 OS X 中别名为 `moved` .

Event: 'moved' *OS X*

在窗口移动的时候触发.

Event: 'enter-full-screen'

在的窗口进入全屏状态时候触发.

Event: 'leave-full-screen'

在的窗口退出全屏状态时候触发.

Event: 'enter-html-full-screen'

在的窗口通过 html api 进入全屏状态时候触发.

Event: 'leave-html-full-screen'

在的窗口通过 html api 退出全屏状态时候触发.

Event: 'app-command' *Windows*

在请求一个App Command.aspx)的时候触发. 典型的是键盘媒体或浏览器命令, Windows上的 "Back" 按钮用作鼠标也会触发.

```
someWindow.on('app-command', function(e, cmd) {
  // Navigate the window back when the user hits their mouse back button
  if (cmd === 'browser-backward' && someWindow.webContents.canGoBack()) {
    someWindow.webContents.goBack();
  }
});
```

Event: 'scroll-touch-begin' *OS X*

在滚动条事件开始的时候触发.

Event: 'scroll-touch-end' *OS X*

在滚动条事件结束的时候触发.

方法

`BrowserWindow` 对象有如下方法:

`BrowserWindow.getAllWindows()`

返回一个所有已经打开了窗口的对象数组.

`BrowserWindow.getFocusedWindow()`

返回应用当前获得焦点窗口,如果没有就返回 `null`.

`BrowserWindow.fromWebContents(webContents)`

- `webContents` WebContents

根据 `webContents` 查找窗口.

`BrowserWindow.fromId(id)`

- `id` Integer

根据 id 查找窗口.

`BrowserWindow.addDevToolsExtension(path)`

- `path` String

添加位于 `path` 的开发者工具栏扩展,并且返回扩展项的名字.

这个扩展会被添加到历史，所以只需要使用这个API一次，这个api不可用作编程使用.

`BrowserWindow.removeDevToolsExtension(name)`

- `name` String

删除开发者工具栏名为 `name` 的扩展.

实例属性

使用 `new BrowserWindow` 创建的实例对象，有如下属性:

```
// In this example `win` is our instance
var win = new BrowserWindow({ width: 800, height: 600 });
```

`win.webContents`

这个窗口的 `WebContents` 对象，所有与界面相关的事件和方法都通过它完成的.

查看 `webContents` documentation 的方法和事件.

`win.id`

窗口的唯一id.

实例方法

使用 `new BrowserWindow` 创建的实例对象，有如下方法:

注意: 一些方法只能在特定os环境中调用，已经尽可能地标出.

`win.destroy()`

强制关闭窗口, `unload` and `beforeunload` 不会触发，并且 `close` 也不会触发, 但是它保证了 `closed` 触发.

`win.close()`

尝试关闭窗口，这与用户点击关闭按钮的效果一样. 虽然网页可能会取消关闭，查看 close event.

`win.focus()`

窗口获得焦点.

`win.isFocused()`

返回 boolean, 窗口是否获得焦点.

`win.show()`

展示并且使窗口获得焦点.

`win.showInactive()`

展示窗口但是不获得焦点.

`win.hide()`

隐藏窗口.

`win.isVisible()`

返回 boolean, 窗口是否可见.

`win.maximize()`

窗口最大化.

`win.unmaximize()`

取消窗口最大化.

`win.isMaximized()`

返回 boolean, 窗口是否最大化.

`win.minimize()`

窗口最小化. 在一些os中，它将在dock中显示.

`win.restore()`

将最小化的窗口恢复为之前的状态.

`win.isMinimized()`

返回 boolean, 窗口是否最小化.

`win.setFullScreen(flag)`

- `flag` Boolean

设置是否全屏.

`win.isFullScreen()`

返回 boolean, 窗口是否全屏化.

`win.setAspectRatio(aspectRatio[, extraSize])` *OS X*

- `aspectRatio` 维持部分视图内容窗口的高宽比值.
- `extraSize` Object (可选) - 维持高宽比值时不包含的额外size.
    - `width` Integer
    - `height` Integer

由一个窗口来维持高宽比值. `extraSize` 允许开发者使用它，它的单位为像素，不包含在 `aspectRatio` 中.这个 API 可用来区分窗口的size和内容的size .

想象一个普通可控的HD video 播放器窗口. 假如左边缘有15控制像素，右边缘有25控制像素，在播放器下面有50控制像素.为了在播放器内保持一个 16:9 的高宽比例，我们可以调用这个api传入参数16/9 and [40, 50].第二个参数不管网页中的额外的宽度和高度在什么位置，只要它们存在就行.只需要把网页中的所有额外的高度和宽度加起来就行.

`win.setBounds(options[, animate])`

- `options` Object
    - `x` Integer
    - `y` Integer
    - `width` Integer
    - `height` Integer
- `animate` Boolean (可选) *OS X*

重新设置窗口的宽高值，并且移动到指定的 `x`, `y` 位置.

`win.getBounds()`

返回一个对象，它包含了窗口的宽，高，x坐标，y坐标.

`win.setSize(width, height[, animate])`

- `width` Integer
- `height` Integer
- `animate` Boolean (可选) *OS X*

重新设置窗口的宽高值.

`win.getSize()`

返回一个数组，它包含了窗口的宽，高.

`win.setContentSize(width, height[, animate])`

- `width` Integer
- `height` Integer
- `animate` Boolean (可选) *OS X*

重新设置窗口客户端的宽高值（例如网页界面）.

`win.getContentSize()`

返回一个数组，它包含了窗口客户端的宽，高.

`win.setMinimumSize(width, height)`

- `width` Integer
- `height` Integer

设置窗口最小化的宽高值.

`win.getMinimumSize()`

返回一个数组，它包含了窗口最小化的宽，高.

`win.setMaximumSize(width, height)`

- `width` Integer
- `height` Integer

设置窗口最大化的宽高值.

`win.getMaximumSize()`

返回一个数组，它包含了窗口最大化的宽，高.

`win.setResizable(resizable)`

- `resizable` Boolean

设置窗口是否可以被用户改变size.

`win.isResizable()`

返回 boolean,窗口是否可以被用户改变size.

`win.setMovable(movable)` *OS X Windows*

- `movable` Boolean

设置窗口是否可以被用户拖动. Linux 无效.

`win.isMovable()` *OS X Windows*

返回 boolean,窗口是否可以被用户拖动. Linux 总是返回 `true` .

`win.setMinimizable(minimizable)` *OS X Windows*

- `minimizable` Boolean

设置窗口是否可以最小化. Linux 无效.

`win.isMinimizable()` *OS X Windows*

返回 boolean,窗口是否可以最小化. Linux 总是返回 `true` .

`win.setMaximizable(maximizable)` *OS X Windows*

- `maximizable` Boolean

设置窗口是否可以最大化. Linux 无效.

`win.isMaximizable()` *OS X Windows*

返回 boolean,窗口是否可以最大化. Linux 总是返回 `true` .

`win.setFullScreenable(fullscreenable)`

- `fullscreenable` Boolean

设置点击最大化按钮是否可以全屏或最大化窗口.

`win.isFullScreenable()`

返回 boolean,点击最大化按钮是否可以全屏或最大化窗口.

`win.setClosable(closable)` *OS X Windows*

- `closable` Boolean

设置窗口是否可以人为关闭. Linux 无效.

`win.isClosable()` *OS X Windows*

返回 boolean,窗口是否可以人为关闭. Linux 总是返回 `true` .

`win.setAlwaysOnTop(flag)`

- `flag` Boolean

是否设置这个窗口始终在其他窗口之上.设置之后，这个窗口仍然是一个普通的窗口，不是一个不可以获得焦点的工具箱窗口.

`win.isAlwaysOnTop()`

返回 boolean,当前窗口是否始终在其它窗口之前.

`win.center()`

窗口居中.

`win.setPosition(x, y[, animate])`

- `x` Integer
- `y` Integer
- `animate` Boolean (可选) *OS X*

移动窗口到对应的 `x` and `y` 坐标.

`win.getPosition()`

返回一个包含当前窗口位置的数组.

`win.setTitle(title)`

- `title` String

改变原窗口的title.

`win.getTitle()`

返回原窗口的title.

注意: 界面title可能和窗口title不相同.

`win.flashFrame(flag)`

- `flag` Boolean

开始或停止显示窗口来获得用户的关注.

`win.setSkipTaskbar(skip)`

- `skip` Boolean

让窗口不在任务栏中显示.

`win.setKiosk(flag)`

- `flag` Boolean

进入或离开 kiosk 模式.

`win.isKiosk()`

返回 boolean,是否进入或离开 kiosk 模式.

`win.getNativeWindowHandle()`

以 `Buffer` 形式返回这个具体平台的窗口的句柄.

windows上句柄类型为 `HWND` ，OS X `NSView*` ， Linux `Window` .

`win.hookWindowMessage(message, callback)` *Windows*

- `message` Integer
- `callback` Function

拦截windows 消息，在 WndProc 接收到消息时触发 `callback` 函数.

`win.isWindowMessageHooked(message)` *Windows*

- `message` Integer

返回 `true` or `false` 来代表是否拦截到消息.

`win.unhookWindowMessage(message)` *Windows*

- `message` Integer

不拦截窗口消息.

`win.unhookAllWindowMessages()` *Windows*

窗口消息全部不拦截.

`win.setRepresentedFilename(filename)` *OS X*

- `filename` String

设置窗口当前文件路径，并且将这个文件的图标放在窗口标题栏上.

`win.getRepresentedFilename()` *OS X*

获取窗口当前文件路径.

`win.setDocumentEdited(edited)` *OS X*

- `edited` Boolean

明确指出窗口文档是否可以编辑，如果可以编辑则将标题栏的图标变成灰色.

`win.isDocumentEdited()` *OS X*

返回 boolean,当前窗口文档是否可编辑.

`win.focusOnWebView()`

`win.blurWebView()`

`win.capturePage([rect, ]callback)`

- `rect` Object (可选) - 捕获Page位置
  - `x` Integer
  - `y` Integer
  - `width` Integer
  - `height` Integer
- `callback` Function

捕获 `rect` 中的page 的快照.完成后将调用回调函数 `callback` 并返回 `image` . `image` 是存储了快照信息的NativeImage实例.如果不设置 `rect` 则将捕获所有可见page.

`win.print([options])`

类似 `webContents.print([options])`

`win.printToPDF(options, callback)`

类似 `webContents.printToPDF(options, callback)`

`win.loadURL(url[, options])`

类似 `webContents.loadURL(url[, options])` .

`win.reload()`

类似 `webContents.reload` .

`win.setMenu(menu)` *Linux Windows*

- `menu` Menu

设置菜单栏的 `menu` ，设置它为 `null` 则表示不设置菜单栏.

`win.setProgressBar(progress)`

- `progress` Double

在进度条中设置进度值，有效范围 [0, 1.0].

当进度小于0时则不显示进度; 当进度大于0时显示结果不确定.

在libux上，只支持Unity桌面环境，需要指明 `*.desktop` 文件并且在 `package.json` 中添加文件名字.默认它为 `app.getName().desktop` .

`win.setOverlayIcon(overlay, description)` *Windows 7+*

- `overlay` NativeImage - 在底部任务栏右边显示图标.
- `description` String - 描述.

向当前任务栏添加一个 16 x 16 像素的图标，通常用来覆盖一些应用的状态，或者直接来提示用户.

`win.setHasShadow(hasShadow)` *OS X*

- `hasShadow` (Boolean)

设置窗口是否应该有阴影.在Windows和Linux系统无效.

`win.hasShadow()` *OS X*

返回 boolean,设置窗口是否有阴影.在Windows和Linux系统始终返回 `true` .

`win.setThumbarButtons(buttons)` *Windows 7+*

- `buttons` Array

在窗口的任务栏button布局出为缩略图添加一个有特殊button的缩略图工具栏. 返回一个 `Boolean` 对象来指示是否成功添加这个缩略图工具栏.

因为空间有限，缩略图工具栏上的 button 数量不应该超过7个.一旦设置了，由于平台限制，就不能移动它了.但是你可使用一个空数组来调用api来清除 buttons .

所有 `buttons` 是一个 `Button` 对象数组:

- `Button` Object
  - `icon` NativeImage - 在工具栏上显示的图标.
  - `click` Function
  - `tooltip` String (可选) - tooltip 文字.
  - `flags` Array (可选) - 控制button的状态和行为. 默认它是 `['enabled']` .

`flags` 是一个数组，它包含下面这些 `String` s:

- `enabled` - button 为激活状态并且开放给用户.
- `disabled` -button 不可用. 目前它有一个可见的状态来表示它不会响应你的行为.
- `dismissonclick` - 点击button，这个缩略窗口直接关闭.
- `nobackground` - 不绘制边框，仅仅使用图像.
- `hidden` - button 对用户不可见.
- `noninteractive` - button 可用但是不可响应; 也不显示按下的状态. 它的值意味着这是一个在通知单使用 button 的实例.

`win.showDefinitionForSelection()` *OS X*

在界面查找选中文字时显示弹出字典.

`win.setAutoHideMenuBar(hide)`

- `hide` Boolean

设置窗口的菜单栏是否可以自动隐藏. 一旦设置了，只有当用户按下 `Alt` 键时则显示.

如果菜单栏已经可见，调用 `setAutoHideMenuBar(true)` 则不会立刻隐藏.

`win.isMenuBarAutoHide()`

返回 boolean,窗口的菜单栏是否可以自动隐藏.

`win.setMenuBarVisibility(visible)`

- `visible` Boolean

设置菜单栏是否可见.如果菜单栏自动隐藏，用户仍然可以按下 `Alt` 键来显示.

`win.isMenuBarVisible()`

返回 boolean,菜单栏是否可见.

`win.setVisibleOnAllWorkspaces(visible)`

- `visible` Boolean

设置窗口是否在所有地方都可见.

注意: 这个api 在windows无效.

`win.isVisibleOnAllWorkspaces()`

返回 boolean,窗口是否在所有地方都可见.

注意: 在 windows上始终返回 false.

`win.setIgnoreMouseEvents(ignore)` *OS X*

- `ignore` Boolean

忽略窗口的所有鼠标事件.

contentTracing

`content-tracing` 模块是用来收集由底层的Chromium content 模块 产生的搜索数据. 这个模块不具备web接口，所有需要我们在chrome浏览器中添加 `chrome://tracing/` 来加载生成文件从而查看结果.

```
const contentTracing = require('electron').contentTracing;

const options = {
  categoryFilter: '*',
  traceOptions: 'record-until-full,enable-sampling'
}

contentTracing.startRecording(options, function() {
  console.log('Tracing started');

  setTimeout(function() {
    contentTracing.stopRecording('', function(path) {
      console.log('Tracing data recorded to ' + path);
    });
  }, 5000);
});
```

方法

`content-tracing` 模块的方法如下:

`contentTracing.getCategories(callback)`

- `callback` Function

获得一组分类组. 分类组可以更改为新的代码路径。

一旦所有的子进程都接受到了 `getCategories` 方法请求, 分类组将调用 `callback` .

`contentTracing.startRecording(options, callback)`

- `options` Object
  - `categoryFilter` String
  - `traceOptions` String
- `callback` Function

开始向所有进程进行记录.(recording)

一旦收到可以开始记录的请求，记录将会立马启动并且在子进程是异步记录听的. 当所有的子进程都收到 `startRecording` 请求的时候，`callback` 将会被调用.

`categoryFilter` 是一个过滤器，它用来控制那些分类组应该被用来查找.过滤器应当有一个可选的 `-` 前缀来排除匹配的分类组.不允许同一个列表既是包含又是排斥.

例子:

- `test_MyTest*` ,
- `test_MyTest*,test_OtherStuff` ,
- `"-excluded_category1,-excluded_category2`

`traceOptions` 控制着哪种查找应该被启动，这是一个用逗号分隔的列表.可用参数如下:

- `record-until-full`
- `record-continuously`
- `trace-to-console`
- `enable-sampling`
- `enable-systrace`

前3个参数是来查找记录模块，并且以后都互斥.如果在 `traceOptions` 中超过一个跟踪 记录模式，那最后一个的优先级最高.如果没有指明跟踪 记录模式，那么它默认为 `record-until-full` .

在 `traceOptions` 中的参数被解析应用之前，查找参数初始化默认为 (`record_mode` 设置为 `record-until-full` , `enable_sampling` 和 `enable_systrace` 设置为 `false`).

`contentTracing.stopRecording(resultFilePath, callback)`

- `resultFilePath` String
- `callback` Function

停止对所有子进程的记录.

子进程通常缓存查找数据，并且仅仅将数据截取和发送给主进程.这有利于在通过 IPC 发送查找数据之前减小查找时的运行开销，这样做很有价值.因此，发送查找数据，我们应当异步通知所有子进程来截取任何待查找的数据.

一旦所有子进程接收到了 `stopRecording` 请求，将调用 `callback` ，并且返回一个包含查找数据的文件.

如果 `resultFilePath` 不为空，那么将把查找数据写入其中，否则写入一个临时文件.实际文件路径如果不为空，则将调用 `callback` .

`contentTracing.startMonitoring(options, callback)`

- `options` Object
  - `categoryFilter` String
  - `traceOptions` String

- `callback` Function

开始向所有进程进行监听.(monitoring)

一旦收到可以开始监听的请求，记录将会立马启动并且在子进程是异步记监听的. 当所有的子进程都收到 `startMonitoring` 请求的时候，`callback` 将会被调用.

`contentTracing.stopMonitoring(callback)`

- `callback` Function

停止对所有子进程的监听.

一旦所有子进程接收到了 `stopMonitoring` 请求，将调用 `callback` .

`contentTracing.captureMonitoringSnapshot(resultFilePath, cal`

- `resultFilePath` String
- `callback` Function

获取当前监听的查找数据.

子进程通常缓存查找数据，并且仅仅将数据截取和发送给主进程.因为如果直接通过 IPC 来发送查找数据的代价昂贵，我们宁愿避免不必要的查找运行开销.因此，为了停止查找，我们应当异步通知所有子进程来截取任何待查找的数据.

一旦所有子进程接收到了 `captureMonitoringSnapshot` 请求，将调用 `callback` ，并且返回一个包含查找数据的文件.

`contentTracing.getTraceBufferUsage(callback)`

- `callback` Function

通过查找 buffer 进程来获取百分比最大使用量.当确定了TraceBufferUsage 的值确定的时候，就调用 `callback` .

`contentTracing.setWatchEvent(categoryName, eventName, callba`

- `categoryName` String
- `eventName` String
- `callback` Function

任意时刻在任何进程上指定事件发生时将调用 `callback` .

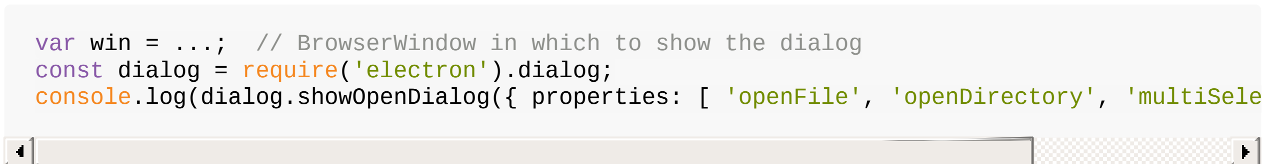
`contentTracing.cancelWatchEvent()`

取消 watch 事件. 如果启动查找，这或许会造成 watch 事件的回调函数 出错.

dialog

`dialog` 模块提供了api来展示原生的系统对话框，例如打开文件框，alert框，所以web应用可以给用户带来跟系统应用相同的体验.

对话框例子，展示了选择文件和目录:



```
var win = ...;  // BrowserWindow in which to show the dialog
const dialog = require('electron').dialog;
console.log(dialog.showOpenDialog({ properties: [ 'openFile', 'openDirectory', 'multiSele
```

OS X 上的注意事项: 如果你想像sheets一样展示对话框，只需要在 `browserWindow` 参数中提供一个 `BrowserWindow` 的引用对象.

方法

`dialog` 模块有以下方法:

`dialog.showOpenDialog([browserWindow, ]options[, callback])`

- `browserWindow` BrowserWindow (可选)
- `options` Object
  - `title` String
  - `defaultPath` String
  - `filters` Array
  - `properties` Array - 包含了对话框的特性值, 可以包含 `openFile` , `openDirectory` , `multiSelections` and `createDirectory`
- `callback` Function (可选)

成功使用这个方法的话，就返回一个可供用户选择的文件路径数组，失败返回 `undefined` .

`filters` 当需要限定用户的行为的时候，指定一个文件数组给用户展示或选择. 例如:

```
{
  filters: [
    { name: 'Images', extensions: ['jpg', 'png', 'gif'] },
    { name: 'Movies', extensions: ['mkv', 'avi', 'mp4'] },
    { name: 'Custom File Type', extensions: ['as'] },
    { name: 'All Files', extensions: ['*'] }
  ]
}
```

`extensions` 数组应当只包含扩展名，不应该包含通配符或'.'号 (例如 `'png'` 正确，但是 `'.png'` 和 `'*.png'` 不正确). 展示全部文件的话, 使用 `'*'` 通配符 (不支持其他通配符).

如果 `callback` 被调用, 将异步调用 API ，并且结果将用过 `callback(filenames)` 展示.

注意: 在 Windows 和 Linux ，一个打开的 dialog 不能既是文件选择框又是目录选择框, 所以如果在这些平台上设置 `properties` 的值为 `['openFile', 'openDirectory']` , 将展示一个目录选择框.

`dialog.showSaveDialog([browserWindow, ]options[, callback])`

- `browserWindow` BrowserWindow (可选)
- `options` Object
  - `title` String
  - `defaultPath` String
  - `filters` Array
- `callback` Function (可选)

成功使用这个方法的话，就返回一个可供用户选择的文件路径数组，失败返回 `undefined`.

`filters` 指定展示一个文件类型数组, 例子 `dialog.showOpenDialog` .

如果 `callback` 被调用, 将异步调用 API ，并且结果将用过 `callback(filenames)` 展示.

`dialog.showMessageBox([browserWindow, ]options[, callback])`

- `browserWindow` BrowserWindow (可选)
- `options` Object
  - `type` String - 可以是 `"none"`, `"info"`, `"error"`, `"question"` 或 `"warning"`. 在 Windows, "question" 与 "info" 展示图标相同, 除非你使用 "icon" 参数.
  - `buttons` Array - buttons 内容，数组.
  - `defaultId` Integer - 在message box 对话框打开的时候，设置默认button选中，值为在 buttons 数组中的button索引.
  - `title` String - message box 的标题，一些平台不显示.
  - `message` String - message box 内容.
  - `detail` String - 额外信息.
  - `icon` NativeImage
  - `cancelId` Integer - 当用户关闭对话框的时候，不是通过点击对话框的button，就返回值.默认值为对应 "cancel" 或 "no" 标签button 的索引值, 或者如果没有这种button，就返回0. 在 OS X 和 Windows 上， "Cancel" button 的索引值将一直是 `cancelId`, 不管之前是不是特别指出的.
  - `noLink` Boolean - 在 Windows ，Electron 将尝试识别哪个button 是普通 button (如 "Cancel" 或 "Yes"), 然后再对话框中以链接命令(command links)方式展现其它的

button . 这能让对话框展示得很炫酷.如果你不喜欢这种效果，你可以设置 `noLink` 为 `true` .

- `callback` Function

展示 message box, 它会阻塞进程，直到 message box 关闭为止.返回点击按钮的索引值.

如果 `callback` 被调用, 将异步调用 API ，并且结果将用过 `callback(response)` 展示.

`dialog.showErrorBox(title, content)`

展示一个传统的包含错误信息的对话框.

在 `app` 模块触发 `ready` 事件之前，这个 api 可以被安全调用，通常它被用来在启动的早期阶段报告错误. 在 Linux 上，如果在 `app` 模块触发 `ready` 事件之前调用，message 将会被触发显示stderr，并且没有实际GUI 框显示.

global-shortcut

`global-shortcut` 模块可以便捷的为您设置(注册/注销)各种自定义操作的快捷键.

Note: 使用此模块注册的快捷键是系统全局的(QQ截图那种), 不要在应用模块(app module)响应 `ready` 消息前使用此模块(注册快捷键).

```
var app = require('app');
var globalShortcut = require('global-shortcut');

app.on('ready', function() {
  // Register a 'ctrl+x' shortcut listener.
  var ret = globalShortcut.register('ctrl+x', function() {
    console.log('ctrl+x is pressed');
  })

  if (!ret) {
    console.log('registration failed');
  }

  // Check whether a shortcut is registered.
  console.log(globalShortcut.isRegistered('ctrl+x'));
});

app.on('will-quit', function() {
  // Unregister a shortcut.
  globalShortcut.unregister('ctrl+x');

  // Unregister all shortcuts.
  globalShortcut.unregisterAll();
});
```

Methods

`global-shortcut` 模块包含以下函数:

`globalShortcut.register(accelerator, callback)`

- `accelerator` Accelerator
- `callback` Function

注册 `accelerator` 快捷键. 当用户按下注册的快捷键时将会调用 `callback` 函数.

`globalShortcut.isRegistered(accelerator)`

- `accelerator` Accelerator

查询 `accelerator` 快捷键是否已经被注册过了,将会返回 `true` (已被注册) 或 `false` (未注册).

`globalShortcut.unregister(accelerator)`

- `accelerator` Accelerator

注销全局快捷键 `accelerator` .

`globalShortcut.unregisterAll()`

注销本应用注册的所有全局快捷键.

ipcMain

`ipcMain` 模块是类 EventEmitter 的实例.当在主进程中使用它的时候，它控制着由渲染进程(web page)发送过来的异步或同步消息.从渲染进程发送过来的消息将触发事件.

发送消息

同样也可以从主进程向渲染进程发送消息，查看更多 webContents.send .

- 发送消息，事件名为 `channel` .
- 回应同步消息, 你可以设置 `event.returnValue` .
- 回应异步消息, 你可以使用 `event.sender.send(...)` .

一个例子，在主进程和渲染进程之间发送和处理消息:

```
// In main process.
const ipcMain = require('electron').ipcMain;
ipcMain.on('asynchronous-message', function(event, arg) {
  console.log(arg);  // prints "ping"
  event.sender.send('asynchronous-reply', 'pong');
});

ipcMain.on('synchronous-message', function(event, arg) {
  console.log(arg);  // prints "ping"
  event.returnValue = 'pong';
});
```

```
// In renderer process (web page).
const ipcRenderer = require('electron').ipcRenderer;
console.log(ipcRenderer.sendSync('synchronous-message', 'ping')); // prints "pong"

ipcRenderer.on('asynchronous-reply', function(event, arg) {
  console.log(arg); // prints "pong"
});
ipcRenderer.send('asynchronous-message', 'ping');
```

监听消息

`ipcMain` 模块有如下监听事件方法:

`ipcMain.on(channel, listener)`

- `channel` String
- `listener` Function

监听 `channel` , 当新消息到达，将通过 `listener(event, args...)` 调用 `listener` .

`ipcMain.once(channel, listener)`

- `channel` String
- `listener` Function

为事件添加一次性用的 `listener` 函数.这个 `listener` 只有在下次的消息到达 `channel` 时被请求调用，之后就被删除了.

`ipcMain.removeListener(channel, listener)`

- `channel` String
- `listener` Function

为特定的 `channel` 从监听队列中删除特定的 `listener` 监听者.

`ipcMain.removeAllListeners([channel])`

- `channel` String (可选)

删除所有监听者，或特指的 `channel` 的所有监听者.

事件对象

传递给 `callback` 的 `event` 对象有如下方法:

`event.returnValue`

将此设置为在一个同步消息中返回的值.

`event.sender`

返回发送消息的 `webContents` ，你可以调用 `event.sender.send` 来回复异步消息，更多信息 webContents.send.

菜单

`menu` 类可以用来创建原生菜单，它可用作应用菜单和 context 菜单.

这个模块是一个主进程的模块，并且可以通过 `remote` 模块给渲染进程调用.

每个菜单有一个或几个菜单项 menu items，并且每个菜单项可以有子菜单.

下面这个例子是在网页(渲染进程)中通过 remote 模块动态创建的菜单，并且右键显示:

```
<!-- index.html -->
<script>
const remote = require('electron').remote;
const Menu = remote.Menu;
const MenuItem = remote.MenuItem;

var menu = new Menu();
menu.append(new MenuItem({ label: 'MenuItem1', click: function() { console.log('item 1 cl
menu.append(new MenuItem({ type: 'separator' }));
menu.append(new MenuItem({ label: 'MenuItem2', type: 'checkbox', checked: true }));

window.addEventListener('contextmenu', function (e) {
  e.preventDefault();
  menu.popup(remote.getCurrentWindow());
}, false);
</script>
```

例子，在渲染进程中使用模板api创建应用菜单:

```
var template = [
  {
    label: 'Edit',
    submenu: [
      {
        label: 'Undo',
        accelerator: 'CmdOrCtrl+Z',
        role: 'undo'
      },
      {
        label: 'Redo',
        accelerator: 'Shift+CmdOrCtrl+Z',
        role: 'redo'
      },
      {
        type: 'separator'
      },
      {
        label: 'Cut',
        accelerator: 'CmdOrCtrl+X',
        role: 'cut'
      },
      {
        label: 'Copy',
        accelerator: 'CmdOrCtrl+C',
        role: 'copy'
      },
      {
        label: 'Paste',
        accelerator: 'CmdOrCtrl+V',
```

```
        role: 'paste'
      },
      {
        label: 'Select All',
        accelerator: 'CmdOrCtrl+A',
        role: 'selectall'
      },
    ]
  },
  {
    label: 'View',
    submenu: [
      {
        label: 'Reload',
        accelerator: 'CmdOrCtrl+R',
        click: function(item, focusedWindow) {
          if (focusedWindow)
            focusedWindow.reload();
        }
      },
      {
        label: 'Toggle Full Screen',
        accelerator: (function() {
          if (process.platform == 'darwin')
            return 'Ctrl+Command+F';
          else
            return 'F11';
        })(),
        click: function(item, focusedWindow) {
          if (focusedWindow)
            focusedWindow.setFullScreen(!focusedWindow.isFullScreen());
        }
      },
      {
        label: 'Toggle Developer Tools',
        accelerator: (function() {
          if (process.platform == 'darwin')
            return 'Alt+Command+I';
          else
            return 'Ctrl+Shift+I';
        })(),
        click: function(item, focusedWindow) {
          if (focusedWindow)
            focusedWindow.toggleDevTools();
        }
      },
    ]
  },
  {
    label: 'Window',
    role: 'window',
    submenu: [
      {
        label: 'Minimize',
        accelerator: 'CmdOrCtrl+M',
        role: 'minimize'
      },
      {
        label: 'Close',
        accelerator: 'CmdOrCtrl+W',
        role: 'close'
      },
    ]
  },
  {
    label: 'Help',
    role: 'help',
    submenu: [
      {
        label: 'Learn More',
        click: function() { require('electron').shell.openExternal('http://electron.atom.
      },
```

```
      ]
    },
  ];

  if (process.platform == 'darwin') {
    var name = require('electron').remote.app.getName();
    template.unshift({
      label: name,
      submenu: [
        {
          label: 'About ' + name,
          role: 'about'
        },
        {
          type: 'separator'
        },
        {
          label: 'Services',
          role: 'services',
          submenu: []
        },
        {
          type: 'separator'
        },
        {
          label: 'Hide ' + name,
          accelerator: 'Command+H',
          role: 'hide'
        },
        {
          label: 'Hide Others',
          accelerator: 'Command+Alt+H',
          role: 'hideothers'
        },
        {
          label: 'Show All',
          role: 'unhide'
        },
        {
          type: 'separator'
        },
        {
          label: 'Quit',
          accelerator: 'Command+Q',
          click: function() { app.quit(); }
        },
      ]
    });
    // Window menu.
    template[3].submenu.push(
      {
        type: 'separator'
      },
      {
        label: 'Bring All to Front',
        role: 'front'
      }
    );
  }

  var menu = Menu.buildFromTemplate(template);
  Menu.setApplicationMenu(menu);
```

类: Menu

`new Menu()`

创建一个新的菜单.

方法

`菜单` 类有如下方法:

`Menu.setApplicationMenu(menu)`

* `menu` Menu

在 OS X 上设置应用菜单 `menu` . 在windows 和 linux，是为每个窗口都在其顶部设置菜单 `menu` .

`Menu.sendActionToFirstResponder(action)` *OS X*

* `action` String

发送 `action` 给应用的第一个响应器.这个用来模仿 Cocoa 菜单的默认行为，通常你只需要使用 `MenuItem` 的属性 `role` .

查看更多 OS X 的原生 action OS X Cocoa Event Handling Guide .

`Menu.buildFromTemplate(template)`

* `template` Array

一般来说， `template` 只是用来创建 MenuItem 的数组 `参数` .

你也可以向 `template` 元素添加其它东西，并且他们会变成已经有的菜单项的属性.

实例方法

`menu` 对象有如下实例方法

`menu.popup([browserWindow, x, y, positioningItem])`

* `browserWindow` BrowserWindow (可选) - 默认为 `null` .
* `x` Number (可选) - 默认为 -1.
* `y` Number (必须 如果x设置了) - 默认为 -1.
* `positioningItem` Number (可选) *OS X* - 在指定坐标鼠标位置下面的菜单项的索引. 默认为 -1.

在 `browserWindow` 中弹出 context menu .你可以选择性地提供指定的 `x, y` 来设置菜单应该放在哪里,否则它将默认地放在当前鼠标的位置.

`menu.append(menuItem)`

- `menuItem` MenuItem

添加菜单项.

`menu.insert(pos, menuItem)`

- `pos` Integer
- `menuItem` MenuItem

在制定位置添加菜单项.

`menu.items()`

获取一个菜单项数组.

OS X Application 上的菜单的注意事项

相对于windows 和 linux, OS X 上的应用菜单是完全不同的style，这里是一些注意事项，来让你的菜单项更原生化.

标准菜单

在 OS X 上，有很多系统定义的标准菜单，例如 `Services` and `Windows` 菜单.为了让你的应用更标准化，你可以为你的菜单的 `role` 设置值，然后 electron 将会识别他们并且让你的菜单更标准:

- `window`
- `help`
- `services`

标准菜单项行为

OS X 为一些菜单项提供了标准的行为方法，例如 `About xxx` , `Hide xxx` , and `Hide Others` . 为了让你的菜单项的行为更标准化，你应该为菜单项设置 `role` 属性.

主菜单名

在 OS X ，无论你设置的什么标签，应用菜单的第一个菜单项的标签始终未你的应用名字.想要改变它的话，你必须通过修改应用绑定的 `Info.plist` 文件来修改应用名字.更多信息参考About Information Property List Files .

为制定浏览器窗口设置菜单 (*Linux Windows*)

浏览器窗口的[`setMenu` 方法][setMenu] 能够设置菜单为特定浏览器窗口的类型.

菜单项位置

当通过 `Menu.buildFromTemplate` 创建菜单的时候，你可以使用 `position` and `id` 来放置菜单项.

`MenuItem` 的属性 `position` 格式为 `[placement]=[id]` ， `placement` 取值为 `before` , `after` , 或 `endof` 和 `id` ， `id` 是菜单已经存在的菜单项的唯一 ID:

- `before` - 在对应引用id菜单项之前插入. 如果引用的菜单项不存在，则将其插在菜单末尾.
- `after` - 在对应引用id菜单项之后插入. 如果引用的菜单项不存在，则将其插在菜单末尾.
- `endof` - 在逻辑上包含对应引用id菜单项的集合末尾插入. 如果引用的菜单项不存在, 则将使用给定的id创建一个新的集合，并且这个菜单项将插入.

当一个菜档项插入成功了，所有的没有插入的菜单项将一个接一个地在后面插入.所以如果你想在同一个位置插入一组菜单项，只需要为这组菜单项的第一个指定位置.

例子

模板:

```
[
  {label: '4', id: '4'},
  {label: '5', id: '5'},
  {label: '1', id: '1', position: 'before=4'},
  {label: '2', id: '2'},
  {label: '3', id: '3'}
]
```

菜单:

```
- 1
- 2
- 3
- 4
- 5
```

模板:

```
[
  {label: 'a', position: 'endof=letters'},
  {label: '1', position: 'endof=numbers'},
  {label: 'b', position: 'endof=letters'},
  {label: '2', position: 'endof=numbers'},
  {label: 'c', position: 'endof=letters'},
  {label: '3', position: 'endof=numbers'}
]
```

菜单:

```
- ---
- a
- b
- c
- ---
- 1
- 2
- 3
```

[setMenu]: https://github.com/electron/electron/blob/master/docs/api/browser-window.md#winsetmenumenu-linux-windows

菜单项

菜单项模块允许你向应用或menu添加选项。

查看menu例子。

类：MenuItem

使用下面的方法创建一个新的 `MenuItem`

new MenuItem(options)

- `options` Object
  - `click` Function - 当菜单项被点击的时候，使用 `click(menuItem,browserWindow)` 调用
  - `role` String - 定义菜单项的行为，在指定 `click` 属性时将会被忽略
  - `type` String - 取值 `normal` ，`separator` ，`checkbox` or `radio`
  - `label` String
  - `sublabel` String
  - `accelerator` Accelerator
  - `icon` NativeImage
  - `enabled` Boolean
  - `visible` Boolean
  - `checked` Boolean
  - `submenu` Menu - 应当作为 `submenu` 菜单项的特定类型，当它作为 `type: 'submenu'` 菜单项的特定类型时可以忽略。如果它的值不是 `Menu` ，将自动转为 `Menu.buildFromTemplate` 。
  - `id` String - 标志一个菜单的唯一性。如果被定义使用，它将被用作这个菜单项的参考位置属性。
  - `position` String - 定义给定的菜单的具体指定位置信息。

在创建菜单项时，如果有匹配的方法，建议指定 `role` 属性，不需要人为操作它的行为，这样菜单使用可以给用户最好的体验。

`role` 属性值可以为：

- `undo`
- `redo`
- `cut`
- `copy`

- `paste`
- `selectall`
- `minimize` - 最小化当前窗口
- `close` - 关闭当前窗口

在 OS X 上，`role` 还可以有以下值：

- `about` - 匹配 `orderFrontStandardAboutPanel` 行为
- `hide` - 匹配 `hide` 行为
- `hideothers` - 匹配 `hideOtherApplications` 行为
- `unhide` - 匹配 `unhideAllApplications` 行为
- `front` - 匹配 `arrangeInFront` 行为
- `window` - "Window" 菜单项
- `help` - "Help" 菜单项
- `services` - "Services" 菜单项

powerMonitor

`power-monitor` 模块是用来监听能源区改变的.只能在主进程中使用.在 `app` 模块的 `ready` 事件触发之后就不能使用这个模块了.

例如:

```
app.on('ready', function() {
  require('electron').powerMonitor.on('suspend', function() {
    console.log('The system is going to sleep');
  });
});
```

事件

`power-monitor` 模块可以触发下列事件:

Event: 'suspend'

在系统挂起的时候触发.

Event: 'resume'

在系统恢复继续工作的时候触发. Emitted when system is resuming.

Event: 'on-ac'

在系统使用交流电的时候触发. Emitted when the system changes to AC power.

Event: 'on-battery'

在系统使用电池电源的时候触发. Emitted when system changes to battery power.

powerSaveBlocker

`powerSaveBlocker` 模块是用来阻止应用系统进入睡眠模式的，因此这允许应用保持系统和屏幕继续工作.

例如:

```
const powerSaveBlocker = require('electron').powerSaveBlocker;

var id = powerSaveBlocker.start('prevent-display-sleep');
console.log(powerSaveBlocker.isStarted(id));

powerSaveBlocker.stop(id);
```

方法

`powerSaveBlocker` 模块有如下方法:

`powerSaveBlocker.start(type)`

- `type` String - 强行保存阻塞类型.
  - `prevent-app-suspension` - 阻止应用挂起. 保持系统活跃，但是允许屏幕不亮. 用例: 下载文件或者播放音频.
  - `prevent-display-sleep` - 阻止应用进入休眠. 保持系统和屏幕活跃，屏幕一直亮. 用例: 播放音频.

开始阻止系统进入睡眠模式.返回一个整数，这个整数标识了保持活跃的blocker.

注意: `prevent-display-sleep` 有更高的优先级 `prevent-app-suspension` . 只有最高优先级生效. 换句话说, `prevent-display-sleep` 优先级永远高于 `prevent-app-suspension` .

例如, A 请求调用了 `prevent-app-suspension` , B请求调用了 `prevent-display-sleep` . `prevent-display-sleep` 将一直工作，直到B停止调用. 在那之后, `prevent-app-suspension` 才起效.

`powerSaveBlocker.stop(id)`

- `id` Integer - 通过 `powerSaveBlocker.start` 返回的保持活跃的 blocker id.

让指定blocker 停止活跃.

`powerSaveBlocker.isStarted(id)`

- `id` Integer - 通过 `powerSaveBlocker.start` 返回的保持活跃的 blocker id.

返回 boolean， 是否对应的 `powerSaveBlocker` 已经启动.

协议

`protocol` 模块可以注册一个自定义协议，或者使用一个已经存在的协议.

例子，使用一个与 `file://` 功能相似的协议 :

```
const electron = require('electron');
const app = electron.app;
const path = require('path');

app.on('ready', function() {
    var protocol = electron.protocol;
    protocol.registerFileProtocol('atom', function(request, callback) {
      var url = request.url.substr(7);
      callback({path: path.normalize(__dirname + '/' + url)});
    }, function (error) {
      if (error)
        console.error('Failed to register protocol')
    });
});
```

注意: 这个模块只有在 `app` 模块的 `ready` 事件触发之后才可使用.

方法

`protocol` 模块有如下方法:

`protocol.registerStandardSchemes(schemes)`

- `schemes` Array - 将一个自定义的方案注册为标准的方案.

一个标准的 `scheme` 遵循 RFC 3986 的 generic URI syntax 标准. 这包含了 `file:` 和 `filesystem:` .

`protocol.registerServiceWorkerSchemes(schemes)`

- `schemes` Array - 将一个自定义的方案注册为处理 service workers.

`protocol.registerFileProtocol(scheme, handler[, completion])`

- `scheme` String
- `handler` Function
- `completion` Function (可选)

注册一个协议，用来发送响应文件.当通过这个协议来发起一个请求的时候，将使用 `handler(request, callback)` 来调用 `handler` .当 `scheme` 被成功注册或者完成(错误)时失败，将使用 `completion(null)` 调用 `completion` .

- `request` Object
  - `url` String
  - `referrer` String
  - `method` String
  - `uploadData` Array (可选)
- `callback` Function

`uploadData` 是一个 `data` 对象数组:

- `data` Object
  - `bytes` Buffer - 被发送的内容.
  - `file` String - 上传的文件路径.

为了处理请求，调用 `callback` 时需要使用文件路径或者一个带 `path` 参数的对象, 例如 `callback(filePath)` 或 `callback({path: filePath})` .

当不使用任何参数调用 `callback` 时，你可以指定一个数字或一个带有 `error` 参数的对象，来标识 `request` 失败.你可以使用的 error number 可以参考 net error list.

默认 `scheme` 会被注册为一个 `http:` 协议，它与遵循 "generic URI syntax" 规则的协议解析不同，例如 `file:` ，所以你或许应该调用 `protocol.registerStandardSchemes` 来创建一个标准的 scheme.

`protocol.registerBufferProtocol(scheme, handler[, completion`

- `scheme` String
- `handler` Function
- `completion` Function (可选)

注册一个 `scheme` 协议，用来发送响应 `Buffer` .

这个方法的用法类似 `registerFileProtocol` ，除非使用一个 `Buffer` 对象，或一个有 `data` , `mimeType` , 和 `charset` 属性的对象来调用 `callback` .

例子:

```
protocol.registerBufferProtocol('atom', function(request, callback) {
  callback({mimeType: 'text/html', data: new Buffer('<h5>Response</h5>')});
}, function (error) {
  if (error)
    console.error('Failed to register protocol')
});
```

`protocol.registerStringProtocol(scheme, handler[, completion`

- `scheme` String
- `handler` Function
- `completion` Function (可选)

注册一个 `scheme` 协议，用来发送响应 `String` .

这个方法的用法类似 `registerFileProtocol` ，除非使用一个 `String` 对象，或一个有 `data` , `mimeType` , 和 `charset` 属性的对象来调用 `callback` .

`protocol.registerHttpProtocol(scheme, handler[, completion])`

- `scheme` String
- `handler` Function
- `completion` Function (可选)

注册一个 `scheme` 协议，用来发送 HTTP 请求作为响应.

这个方法的用法类似 `registerFileProtocol` ，除非使用一个 `redirectRequest` 对象，或一个有 `url` , `method` , `referrer` , `uploadData` 和 `session` 属性的对象来调用 `callback` .

- `redirectRequest` Object
  - `url` String
  - `method` String
  - `session` Object (可选)
  - `uploadData` Object (可选)

默认这个 HTTP 请求会使用当前 session .如果你想使用不同的session值，你应该设置 `session` 为 `null` .

POST 请求应当包含 `uploadData` 对象.

- `uploadData` object
  - `contentType` String - 内容的 MIME type.
  - `data` String - 被发送的内容.

`protocol.unregisterProtocol(scheme[, completion])`

- `scheme` String
- `completion` Function (可选)

注销自定义协议 `scheme` .

`protocol.isProtocolHandled(scheme, callback)`

- `scheme` String
- `callback` Function

将使用一个布尔值来调用 `callback` ，这个布尔值标识了是否已经存在 `scheme` 的句柄了.

`protocol.interceptFileProtocol(scheme, handler[, completion]`

- `scheme` String
- `handler` Function
- `completion` Function (可选)

拦截 `scheme` 协议并且使用 `handler` 作为协议的新的句柄来发送响应文件.

`protocol.interceptStringProtocol(scheme, handler[, completio`

- `scheme` String
- `handler` Function
- `completion` Function (可选)

拦截 `scheme` 协议并且使用 `handler` 作为协议的新的句柄来发送响应 `String` .

`protocol.interceptBufferProtocol(scheme, handler[, completio`

- `scheme` String
- `handler` Function
- `completion` Function (可选)

拦截 `scheme` 协议并且使用 `handler` 作为协议的新的句柄来发送响应 `Buffer` .

`protocol.interceptHttpProtocol(scheme, handler[, completion]`

- `scheme` String
- `handler` Function
- `completion` Function (optional)

拦截 `scheme` 协议并且使用 `handler` 作为协议的新的句柄来发送新的响应 HTTP 请求. Intercepts `scheme` protocol and uses `handler` as the protocol's new handler which sends a new HTTP request as a response.

`protocol.uninterceptProtocol(scheme[, completion])`

- `scheme` String
- `completion` Function 取消对 `scheme` 的拦截，使用它的原始句柄进行处理.

session

`session` 模块可以用来创建一个新的 `Session` 对象.

你也可以通过使用 `webContents` 的属性 `session` 来使用一个已有页面的 `session` ，`webContents` 是 `BrowserWindow` 的属性.

```
const BrowserWindow = require('electron').BrowserWindow;

var win = new BrowserWindow({ width: 800, height: 600 });
win.loadURL("http://github.com");

var ses = win.webContents.session;
```

方法

`session` 模块有如下方法:

session.fromPartition(partition)

- `partition` String

从字符串 `partition` 返回一个新的 `Session` 实例.

如果 `partition` 以 `persist:` 开头，那么这个page将使用一个持久的 session，这个 session 将对应用的所有 page 可用.如果没前缀，这个 page 将使用一个历史 session.如果 `partition` 为空，那么将返回应用的默认 session .

属性

`session` 模块有如下属性:

session.defaultSession

返回应用的默认 session 对象.

Class: Session

可以在 `session` 模块中创建一个 `Session` 对象 :

```
const session = require('electron').session;

var ses = session.fromPartition('persist:name');
```

实例事件

实例 `Session` 有以下事件:

Event: 'will-download'

- `event` Event
- `item` DownloadItem
- `webContents` WebContents

当 Electron 将要从 `webContents` 下载 `item` 时触发.

调用 `event.preventDefault()` 可以取消下载，并且在进程的下个 tick中，这个 `item` 也不可用.

```
session.defaultSession.on('will-download', function(event, item, webContents) {
  event.preventDefault();
  require('request')(item.getURL(), function(data) {
    require('fs').writeFileSync('/somewhere', data);
  });
});
```

实例方法

实例 `Session` 有以下方法:

`ses.cookies`

`cookies` 赋予你全力来查询和修改 cookies. 例如:

```
// 查询所有 cookies.
session.defaultSession.cookies.get({}, function(error, cookies) {
  console.log(cookies);
});

// 查询与指定 url 相关的所有 cookies.
session.defaultSession.cookies.get({ url : "http://www.github.com" }, function(error, coo
  console.log(cookies);
});

// 设置 cookie;
// may overwrite equivalent cookies if they exist.
var cookie = { url : "http://www.github.com", name : "dummy_name", value : "dummy" };
session.defaultSession.cookies.set(cookie, function(error) {
  if (error)
    console.error(error);
});
```

`ses.cookies.get(filter, callback)`

- `filter` Object
  - `url` String (可选) - 与获取 cookies 相关的 `url`.不设置的话就是从所有 url 获取 cookies .
  - `name` String (可选) - 通过 name 过滤 cookies.
  - `domain` String (可选) - 获取对应域名或子域名的 cookies .
  - `path` String (可选) - 获取对应路径的 cookies .
  - `secure` Boolean (可选) - 通过安全性过滤 cookies.
  - `session` Boolean (可选) - 过滤掉 session 或 持久的 cookies.
- `callback` Function

发送一个请求，希望获得所有匹配 `details` 的 cookies, 在完成的时候，将通过 `callback(error, cookies)` 调用 `callback`.

`cookies`是一个 `cookie` 对象.

- `cookie` Object
  - `name` String - cookie 名.
  - `value` String - cookie值.
  - `domain` String - cookie域名.
  - `hostOnly` String - 是否 cookie 是一个 host-only cookie.
  - `path` String - cookie路径.
  - `secure` Boolean - 是否是安全 cookie.
  - `httpOnly` Boolean - 是否只是 HTTP cookie.
  - `session` Boolean - cookie 是否是一个 session cookie 或一个带截至日期的持久 cookie .
  - `expirationDate` Double (可选) - cookie的截至日期，数值为UNIX纪元以来的秒数. 对session cookies 不提供.

`ses.cookies.set(details, callback)`

- `details` Object
  - `url` String - 与获取 cookies 相关的 `url` .
  - `name` String - cookie 名. 忽略默认为空.
  - `value` String - cookie 值. 忽略默认为空.
  - `domain` String - cookie的域名. 忽略默认为空.
  - `path` String - cookie 的路径. 忽略默认为空.
  - `secure` Boolean - 是否已经进行了安全性标识. 默认为 false.
  - `session` Boolean - 是否已经 HttpOnly 标识. 默认为 false.
  - `expirationDate` Double - cookie的截至日期，数值为UNIX纪元以来的秒数. 如果忽略, cookie 变为 session cookie.
- `callback` Function

使用 `details` 设置 cookie, 完成时使用 `callback(error)` 掉哟个 `callback` .

`ses.cookies.remove(url, name, callback)`

- `url` String - 与 cookies 相关的 `url`.
- `name` String - 需要删除的 cookie 名.
- `callback` Function

删除匹配 `url` 和 `name` 的 cookie, 完成时使用 `callback()` 调用 `callback` .

`ses.getCacheSize(callback)`

- `callback` Function
  - `size` Integer - 单位 bytes 的缓存 size.

返回 session 的当前缓存 size .

`ses.clearCache(callback)`

- `callback` Function - 操作完成时调用

清空 session 的 HTTP 缓存.

`ses.clearStorageData([options, ]callback)`

- `options` Object (可选)
  - `origin` String - 应当遵循 `window.location.origin` 的格式 `scheme://host:port` .
  - `storages` Array - 需要清理的 storages 类型, 可以包含 : `appcache` , `cookies` , `filesystem` , `indexdb` , `local storage` , `shadercache` , `websql` , `serviceworkers`
  - `quotas` Array - 需要清理的类型指标, 可以包含: `temporary` , `persistent` , `syncable` .
- `callback` Function - 操作完成时调用.

清除 web storages 的数据.

`ses.flushStorageData()`

将没有写入的 DOMStorage 写入磁盘.

`ses.setProxy(config, callback)`

- `config` Object
  - `pacScript` String - 与 PAC 文件相关的 URL.
  - `proxyRules` String - 代理使用规则.
- `callback` Function - 操作完成时调用.

设置 proxy settings.

当 `pacScript` 和 `proxyRules` 一同提供时，将忽略 `proxyRules` ，并且使用 `pacScript` 配置 .

`proxyRules` 需要遵循下面的规则:

```
proxyRules = schemeProxies[";"<schemeProxies>]
schemeProxies = [<urlScheme>"="]<proxyURIList>
urlScheme = "http" | "https" | "ftp" | "socks"
proxyURIList = <proxyURL>[","<proxyURIList>]
proxyURL = [<proxyScheme>"://"]<proxyHost>[":"<proxyPort>]
```

例子:

- `http=foopy:80;ftp=foopy2` - 为 `http://` URL 使用 HTTP 代理 `foopy:80` , 和为 `ftp://` URL HTTP 代理 `foopy2:80` .
- `foopy:80` - 为所有 URL 使用 HTTP 代理 `foopy:80` .
- `foopy:80,bar,direct://` - 为所有 URL 使用 HTTP 代理 `foopy:80` , 如果 `foopy:80` 不可用，则切换使用 `bar` , 再往后就不使用代理了.
- `socks4://foopy` - 为所有 URL 使用 SOCKS v4 代理 `foopy:1080` .
- `http=foopy,socks5://bar.com` - 为所有 URL 使用 HTTP 代理 `foopy` , 如果 `foopy` 不可用，则切换到 SOCKS5 代理 `bar.com` .
- `http=foopy,direct://` - 为所有http url 使用 HTTP 代理，如果 `foopy` 不可用，则不使用代理.
- `http=foopy;socks=foopy2` - 为所有http url 使用 `foopy` 代理，为所有其他 url 使用 `socks4://foopy2` 代理.

`ses.resolveProxy(url, callback)`

- `url` URL
- `callback` Function

解析 `url` 的代理信息.当请求完成的时候使用 `callback(proxy)` 调用 `callback` .

`ses.setDownloadPath(path)`

- `path` String - 下载地址

设置下载保存地址，默认保存地址为各自 app 应用的 `Downloads` 目录.

`ses.enableNetworkEmulation(options)`

- `options` Object
  - `offline` Boolean - 是否模拟网络故障.
  - `latency` Double - 每毫秒的 RTT
  - `downloadThroughput` Double - 每 Bps 的下载速率.
  - `uploadThroughput` Double - 每 Bps 的上载速率.

通过给定配置的 `session` 来模拟网络.

```
// 模拟 GPRS 连接，使用的 50kbps 流量，500 毫秒的 rtt.
window.webContents.session.enableNetworkEmulation({
    latency: 500,
    downloadThroughput: 6400,
    uploadThroughput: 6400
});

// 模拟网络故障.
window.webContents.session.enableNetworkEmulation({offline: true});
```

`ses.disableNetworkEmulation()`

停止所有已经使用 `session` 的活跃模拟网络. 重置为原始网络类型.

`ses.setCertificateVerifyProc(proc)`

- `proc` Function

为 `session` 设置证书验证过程，当请求一个服务器的证书验证时，使用 `proc(hostname, certificate, callback)` 调用 `proc`.调用 `callback(true)` 来接收证书，调用 `callback(false)` 来拒绝验证证书.

调用了 `setCertificateVerifyProc(null)` ，则将会回复到默认证书验证过程.

```
myWindow.webContents.session.setCertificateVerifyProc(function(hostname, cert, callback)
  if (hostname == 'github.com')
    callback(true);
  else
    callback(false);
});
```

`ses.setPermissionRequestHandler(handler)`

- `handler` Function
  - `webContents` Object - WebContents 请求许可.
  - `permission` String - 枚举了 'media', 'geolocation', 'notifications', 'midiSysex', 'pointerLock', 'fullscreen'.
  - `callback` Function - 允许或禁止许可.

为对应 `session` 许可请求设置响应句柄.调用 `callback(true)` 接收许可，调用 `callback(false)` 禁止许可.

```
session.fromPartition(partition).setPermissionRequestHandler(function(webContents, permis
  if (webContents.getURL() === host) {
    if (permission == "notifications") {
      callback(false); // denied.
      return;
    }
  }

  callback(true);
});
```

`ses.clearHostResolverCache([callback])`

- `callback` Function (可选) - 操作结束调用.

清除主机解析缓存.

`ses.webRequest`

在其生命周期的不同阶段，`webRequest` API 设置允许拦截并修改请求内容.

每个 API 接收一可选的 `filter` 和 `listener`，当 API 事件发生的时候使用 `listener(details)` 调用 `listener`，`details` 是一个用来描述请求的对象.为 `listener` 使用 `null` 则会退定事件.

`filter` 是一个拥有 `urls` 属性的对象，这是一个 url 模式数组，这用来过滤掉不匹配指定 url 模式的请求.如果忽略 `filter` ，那么所有请求都将可以成功匹配.

所有事件的 `listener` 都有一个回调事件，当 `listener` 完成它的工作的时候，它将使用一个 `response` 对象来调用.

```
// 将所有请求的代理都修改为下列 url.
var filter = {
  urls: ["https://*.github.com/*", "*://electron.github.io"]
};

session.defaultSession.webRequest.onBeforeSendHeaders(filter, function(details, callback)
  details.requestHeaders['User-Agent'] = "MyAgent";
  callback({cancel: false, requestHeaders: details.requestHeaders});
});
```

`ses.webRequest.onBeforeRequest([filter, ]listener)`

- `filter` Object
- `listener` Function

当一个请求即将开始的时候，使用 `listener(details, callback)` 调用 `listener`.

- `details` Object
  - `id` Integer
  - `url` String

  - `method` String
  - `resourceType` String
  - `timestamp` Double
  - `uploadData` Array (可选)
- `callback` Function

`uploadData` 是一个 `data` 数组对象:

- `data` Object
  - `bytes` Buffer - 被发送的内容.
  - `file` String - 上载文件路径.

`callback` 必须使用一个 `response` 对象来调用:

- `response` Object
  - `cancel` Boolean (可选)
  - `redirectURL` String (可选) - 原始请求阻止发送或完成，而不是重定向.

`ses.webRequest.onBeforeSendHeaders([filter, ]listener)`

- `filter` Object
- `listener` Function

一旦请求报文头可用了,在发送 HTTP 请求的之前，使用 `listener(details, callback)` 调用 `listener`.这也许会在服务器发起一个tcp 连接，但是在发送任何 http 数据之前发生.

- `details` Object
  - `id` Integer
  - `url` String
  - `method` String
  - `resourceType` String
  - `timestamp` Double
  - `requestHeaders` Object
- `callback` Function

必须使用一个 `response` 对象来调用 `callback` :

- `response` Object
  - `cancel` Boolean (可选)
  - `requestHeaders` Object (可选) - 如果提供了,将使用这些 headers 来创建请求.

`ses.webRequest.onSendHeaders([filter, ]listener)`

- `filter` Object
- `listener` Function

在一个请求正在发送到服务器的时候，使用 `listener(details)` 来调用 `listener` ，之前 `onBeforeSendHeaders` 修改部分响应可用，同时取消监听.

- `details` Object
  - `id` Integer
  - `url` String
  - `method` String
  - `resourceType` String
  - `timestamp` Double
  - `requestHeaders` Object

`ses.webRequest.onHeadersReceived([filter,] listener)`

- `filter` Object
- `listener` Function

当 HTTP 请求报文头已经到达的时候，使用 `listener(details, callback)` 调用 `listener` .

- `details` Object
  - `id` String
  - `url` String
  - `method` String
  - `resourceType` String
  - `timestamp` Double
  - `statusLine` String
  - `statusCode` Integer
  - `responseHeaders` Object
- `callback` Function

必须使用一个 `response` 对象来调用 `callback` :

- `response` Object
  - `cancel` Boolean
  - `responseHeaders` Object (可选) - 如果提供, 服务器将假定使用这些头来响应.

`ses.webRequest.onResponseStarted([filter, ]listener)`

- `filter` Object
- `listener` Function

当响应body的首字节到达的时候，使用 `listener(details)` 调用 `listener`.对 http 请求来说，这意味着状态线和响应头可用了.

- `details` Object
  - `id` Integer

  - `url` String
  - `method` String
  - `resourceType` String
  - `timestamp` Double
  - `responseHeaders` Object
  - `fromCache` Boolean - 标识响应是否来自磁盘 cache.
  - `statusCode` Integer
  - `statusLine` String

`ses.webRequest.onBeforeRedirect([filter, ]listener)`

- `filter` Object
- `listener` Function

当服务器的重定向初始化正要启动时，使用 `listener(details)` 调用 `listener` .

- `details` Object
  - `id` String
  - `url` String
  - `method` String
  - `resourceType` String
  - `timestamp` Double
  - `redirectURL` String
  - `statusCode` Integer
  - `ip` String (可选) - 请求的真实服务器ip 地址
  - `fromCache` Boolean
  - `responseHeaders` Object

`ses.webRequest.onCompleted([filter, ]listener)`

- `filter` Object
- `listener` Function

当请求完成的时候，使用 `listener(details)` 调用 `listener` .

- `details` Object
  - `id` Integer
  - `url` String
  - `method` String
  - `resourceType` String
  - `timestamp` Double
  - `responseHeaders` Object
  - `fromCache` Boolean
  - `statusCode` Integer

- `statusLine` String

`ses.webRequest.onErrorOccurred([filter, ]listener)`

- `filter` Object
- `listener` Function

当一个错误发生的时候，使用 `listener(details)` 调用 `listener`.

- `details` Object
  - `id` Integer
  - `url` String
  - `method` String
  - `resourceType` String
  - `timestamp` Double
  - `fromCache` Boolean
  - `error` String - 错误描述.

webContents

`webContents` 是一个 事件发出者.

它负责渲染并控制网页，也是 `BrowserWindow` 对象的属性.一个使用 `webContents` 的例子:

```
const BrowserWindow = require('electron').BrowserWindow;

var win = new BrowserWindow({width: 800, height: 1500});
win.loadURL("http://github.com");

var webContents = win.webContents;
```

事件

`webContents` 对象可发出下列事件:

Event: 'did-finish-load'

当导航完成时发出事件，`onload` 事件也完成.

Event: 'did-fail-load'

返回:

- `event` Event
- `errorCode` Integer
- `errorDescription` String
- `validatedURL` String

这个事件类似 `did-finish-load` ，但是是在加载失败或取消加载时发出, 例如， `window.stop()` 请求结束.错误代码的完整列表和它们的含义都可以在 here 找到.

Event: 'did-frame-finish-load'

返回:

- `event` Event
- `isMainFrame` Boolean

当一个 frame 导航完成的时候发出事件.

Event: 'did-start-loading'

当 tab 的spinner 开始 spinning的时候.

Event: 'did-stop-loading'

当 tab 的spinner 结束 spinning的时候.

Event: 'did-get-response-details'

返回:

- `event` Event
- `status` Boolean
- `newURL` String
- `originalURL` String
- `httpResponseCode` Integer
- `requestMethod` String
- `referrer` String
- `headers` Object

当有关请求资源的详细信息可用的时候发出事件. `status` 标识了 socket链接来下载资源.

Event: 'did-get-redirect-request'

返回:

- `event` Event
- `oldURL` String
- `newURL` String
- `isMainFrame` Boolean
- `httpResponseCode` Integer
- `requestMethod` String
- `referrer` String
- `headers` Object

当在请求资源时收到重定向的时候发出事件.

Event: 'dom-ready'

返回:

- `event` Event

当指定 frame 中的 文档加载完成的时候发出事件.

Event: 'page-favicon-updated'

返回:

- `event` Event
- `favicons` Array - Array of URLs

当 page 收到图标 url 的时候发出事件.

Event: 'new-window'

返回:

- `event` Event
- `url` String
- `frameName` String
- `disposition` String - 可为 `default` , `foreground-tab` , `background-tab` , `new-window` 和 `other` .
- `options` Object - 创建新的 `BrowserWindow` 时使用的参数.

当 page 请求打开指定 url 窗口的时候发出事件.这可以是通过 `window.open` 或一个外部连接如 `<a target='_blank'>` 发出的请求.

默认指定 `url` 的 `BrowserWindow` 会被创建.

调用 `event.preventDefault()` 可以用来阻止打开窗口.

Event: 'will-navigate'

返回:

- `event` Event
- `url` String

当用户或 page 想要开始导航的时候发出事件.它可在当 `window.location` 对象改变或用户点击 page 中的链接的时候发生.

当使用 api(如 `webContents.loadURL` 和 `webContents.back`) 以编程方式来启动导航的时候，这个事件将不会发出.

它也不会在页内跳转发生， 例如点击锚链接或更新 `window.location.hash` .使用 `did-navigate-in-page` 事件可以达到目的.

调用 `event.preventDefault()` 可以阻止导航.

Event: 'did-navigate'

返回:

* `event` Event
* `url` String

当一个导航结束时候发出事件.

页内跳转时不会发出这个事件，例如点击锚链接或更新 `window.location.hash`.使用 `did-navigate-in-page` 事件可以达到目的.

Event: 'did-navigate-in-page'

返回:

* `event` Event
* `url` String

当页内导航发生的时候发出事件.

当页内导航发生的时候，page 的url 改变，但是不会跳出界面.例如当点击锚链接时或者 DOM 的 `hashchange` 事件发生.

Event: 'crashed'

当渲染进程崩溃的时候发出事件.

Event: 'plugin-crashed'

返回:

* `event` Event
* `name` String
* `version` String

当插件进程崩溃时候发出事件.

Event: 'destroyed'

当 `webContents` 被删除的时候发出事件.

Event: 'devtools-opened'

当开发者工具栏打开的时候发出事件.

Event: 'devtools-closed'

当开发者工具栏关闭时候发出事件.

Event: 'devtools-focused'

当开发者工具栏获得焦点或打开的时候发出事件.

Event: 'certificate-error'

返回:

- `event` Event
- `url` URL
- `error` String - The error code
- `certificate` Object
  - `data` Buffer - PEM encoded data
  - `issuerName` String
- `callback` Function

当验证证书或 `url` 失败的时候发出事件.

使用方法类似 `app` 的 `certificate-error` 事件.

Event: 'select-client-certificate'

返回:

- `event` Event
- `url` URL
- `certificateList` [Objects]
  - `data` Buffer - PEM encoded data
  - `issuerName` String - Issuer's Common Name
- `callback` Function

当请求客户端证书的时候发出事件.

使用方法类似 `app` 的 `select-client-certificate` 事件.

Event: 'login'

返回:

- `event` Event

- `request` Object
  - `method` String
  - `url` URL
  - `referrer` URL
- `authInfo` Object
  - `isProxy` Boolean
  - `scheme` String
  - `host` String
  - `port` Integer
  - `realm` String
- `callback` Function

当 `webContents` 想做基本验证的时候发出事件.

使用方法类似 the `login` event of `app` .

Event: 'found-in-page'

返回:

- `event` Event
- `result` Object
  - `requestId` Integer
  - `finalUpdate` Boolean - 标识是否还有更多的值可以查看.
  - `activeMatchOrdinal` Integer (可选) - 活动匹配位置
  - `matches` Integer (可选) - 匹配数量.
  - `selectionArea` Object (可选) - 协调首个匹配位置.

当使用 `webContents.findInPage` 进行页内查找并且找到可用值得时候发出事件.

Event: 'media-started-playing'

当媒体开始播放的时候发出事件.

Event: 'media-paused'

当媒体停止播放的时候发出事件.

Event: 'did-change-theme-color'

当page 的主题色时候发出事件.这通常由于引入了一个 meta 标签 :

```
<meta name='theme-color' content='#ff0000'>
```

Event: 'cursor-changed'

返回:

- `event` Event
- `type` String
- `image` NativeImage (可选)
- `scale` Float (可选)

当鼠标的类型发生改变的时候发出事件. `type` 的参数可以是 `default` , `crosshair` , `pointer` , `text` , `wait` , `help` , `e-resize` , `n-resize` , `ne-resize` , `nw-resize` , `s-resize` , `se-resize` , `sw-resize` , `w-resize` , `ns-resize` , `ew-resize` , `nesw-resize` , `nwse-resize` , `col-resize` , `row-resize` , `m-panning` , `e-panning` , `n-panning` , `ne-panning` , `nw-panning` , `s-panning` , `se-panning` , `sw-panning` , `w-panning` , `move` , `vertical-text` , `cell` , `context-menu` , `alias` , `progress` , `nodrop` , `copy` , `none` , `not-allowed` , `zoom-in` , `zoom-out` , `grab` , `grabbing` , `custom` .

如果 `type` 参数值为 `custom` , `image` 参数会在一个 `NativeImage` 中控制自定义鼠标图片, 并且 `scale` 会控制图片的缩放比例.

实例方法

`webContents` 对象有如下的实例方法:

`webContents.loadURL(url[, options])`

- `url` URL
- `options` Object (可选)
  - `httpReferrer` String - A HTTP Referrer url.
  - `userAgent` String - 产生请求的用户代理
  - `extraHeaders` String - 以 "\n" 分隔的额外头

在窗口中加载 `url` , `url` 必须包含协议前缀, 比如 `http://` 或 `file://` . 如果加载想要忽略 http 缓存，可以使用 `pragma` 头来达到目的.

```
const options = {"extraHeaders" : "pragma: no-cache\n"}
webContents.loadURL(url, options)
```

`webContents.downloadURL(url)`

- `url` URL

初始化一个指定 `url` 的资源下载，不导航跳转. `session` 的 `will-download` 事件会触发.

`webContents.getURL()`

返回当前page 的 url.

```
var win = new BrowserWindow({width: 800, height: 600});
win.loadURL("http://github.com");

var currentURL = win.webContents.getURL();
```

`webContents.getTitle()`

返回当前page 的 标题.

`webContents.isLoading()`

返回一个布尔值，标识当前页是否正在加载.

`webContents.isWaitingForResponse()`

返回一个布尔值，标识当前页是否正在等待主要资源的第一次响应.

`webContents.stop()`

停止还为开始的导航.

`webContents.reload()`

重载当前页.

`webContents.reloadIgnoringCache()`

重载当前页，忽略缓存.

`webContents.canGoBack()`

返回一个布尔值，标识浏览器是否能回到前一个page.

`webContents.canGoForward()`

返回一个布尔值，标识浏览器是否能前往下一个page.

`webContents.canGoToOffset(offset)`

- `offset` Integer

返回一个布尔值，标识浏览器是否能前往指定 `offset` 的page.

`webContents.clearHistory()`

清除导航历史.

`webContents.goBack()`

让浏览器回退到前一个page.

`webContents.goForward()`

让浏览器回前往下一个page.

`webContents.goToIndex(index)`

- `index` Integer

让浏览器回前往指定 `index` 的page.

`webContents.goToOffset(offset)`

- `offset` Integer

导航到相对于当前页的偏移位置页.

`webContents.isCrashed()`

渲染进程是否崩溃.

`webContents.setUserAgent(userAgent)`

- `userAgent` String

重写本页用户代理.

`webContents.getUserAgent()`

返回一个 `String` ，标识本页用户代理信息.

`webContents.insertCSS(css)`

- `css` String

为当前页插入css.

`webContents.executeJavaScript(code[, userGesture, callback])`

- `code` String
- `userGesture` Boolean (可选)
- `callback` Function (可选) - 脚本执行完成后调用的回调函数.
    - `result`

评估 page `代码` .

浏览器窗口中的一些 HTML API ，例如 `requestFullScreen` ，只能被用户手势请求.设置 `userGesture` 为 `true` 可以取消这个限制.

`webContents.setAudioMuted(muted)`

- `muted` Boolean

减缓当前也的 audio 的播放速度.

`webContents.isAudioMuted()`

返回一个布尔值，标识当前页是否减缓了 audio 的播放速度.

`webContents.undo()`

执行网页的编辑命令 `undo` .

`webContents.redo()`

执行网页的编辑命令 `redo` .

`webContents.cut()`

执行网页的编辑命令 `cut` .

`webContents.copy()`

执行网页的编辑命令 `copy` .

`webContents.paste()`

执行网页的编辑命令 `paste` .

`webContents.pasteAndMatchStyle()`

执行网页的编辑命令 `pasteAndMatchStyle` .

`webContents.delete()`

执行网页的编辑命令 `delete` .

`webContents.selectAll()`

执行网页的编辑命令 `selectAll` .

`webContents.unselect()`

执行网页的编辑命令 `unselect` .

`webContents.replace(text)`

- `text` String

执行网页的编辑命令 `replace` .

`webContents.replaceMisspelling(text)`

- `text` String

执行网页的编辑命令 `replaceMisspelling` .

`webContents.insertText(text)`

- `text` String

插入 `text` 到获得了焦点的元素.

`webContents.findInPage(text[, options])`

- `text` String - 查找内容, 不能为空.
- `options` Object (可选)
  - `forward` Boolean - 是否向前或向后查找, 默认为 `true` .
  - `findNext` Boolean - 当前操作是否是第一次查找或下一次查找, 默认为 `false` .
  - `matchCase` Boolean - 查找是否区分大小写, 默认为 `false` .
  - `wordStart` Boolean -是否仅以首字母查找. 默认为 `false` .
  - `medialCapitalAsWordStart` Boolean - 是否结合 `wordStart` ,如果匹配是大写字母开头，后面接小写字母或无字母，那么就接受这个词中匹配.接受几个其它的合成词匹

配, 默认为 `false` .

发起请求，在网页中查找所有与 `text` 相匹配的项，并且返回一个 `Integer` 来表示这个请求用的请求Id.这个请求结果可以通过订阅 `found-in-page` 事件来取得.

`webContents.stopFindInPage(action)`

- `action` String - 指定一个行为来接替停止 `webContents.findInPage` 请求.
  - `clearSelection` - 转变为一个普通的 selection.
  - `keepSelection` - 清除 selection.
  - `activateSelection` - 获取焦点并点击 selection node.

使用给定的 `action` 来为 `webContents` 停止任何 `findInPage` 请求.

```
webContents.on('found-in-page', function(event, result) {
  if (result.finalUpdate)
    webContents.stopFindInPage("clearSelection");
});

const requestId = webContents.findInPage("api");
```

`webContents.hasServiceWorker(callback)`

- `callback` Function

检查是否有任何 ServiceWorker 注册了，并且返回一个布尔值，来作为 `callback` 响应的标识.

`webContents.unregisterServiceWorker(callback)`

- `callback` Function

如果存在任何 ServiceWorker ，则全部注销，并且当JS承诺执行行或JS拒绝执行而失败的时候，返回一个布尔值，它标识了相应的 `callback` .

`webContents.print([options])`

- `options` Object (可选)
  - `silent` Boolean - 不需要请求用户的打印设置. 默认为 `false` .
  - `printBackground` Boolean - 打印背景和网页图片. 默认为 `false` .

打印窗口的网页. 当设置 `silent` 为 `false` 的时候，Electron 将使用系统默认的打印机和打印方式来打印.

在网页中调用 `window.print()` 和 调用 `webContents.print({silent: false, printBackground: false})` 具有相同的作用.

注意: 在 Windows, 打印 API 依赖于 `pdf.dll`. 如果你的应用不使用任何的打印, 你可以安全删除 `pdf.dll` 来减少二进制文件的size.

`webContents.printToPDF(options, callback)`

- `options` Object
  - `marginsType` Integer - 指定使用的 margin type. 默认 margin 使用 0, 无 margin 使用 1, 最小化 margin 使用 2.
  - `pageSize` String - 指定生成的PDF文件的page size. 可以是 `A3` , `A4` , `A5` , `Legal` , `Letter` 和 `Tabloid` .
  - `printBackground` Boolean - 是否打印 css 背景.
  - `printSelectionOnly` Boolean - 是否只打印选中的部分.
  - `landscape` Boolean - landscape 为 `true` , portrait 为 `false` .
- `callback` Function

打印窗口的网页为 pdf ，使用 Chromium 预览打印的自定义设置.

完成时使用 `callback(error, data)` 调用 `callback` . `data` 是一个 `Buffer` ，包含了生成的 pdf 数据.

默认，空的 `options` 被视为：

```
{
  marginsType: 0,
  printBackground: false,
  printSelectionOnly: false,
  landscape: false
}
```

```
const BrowserWindow = require('electron').BrowserWindow;
const fs = require('fs');

var win = new BrowserWindow({width: 800, height: 600});
win.loadURL("http://github.com");

win.webContents.on("did-finish-load", function() {
  // Use default printing options
  win.webContents.printToPDF({}, function(error, data) {
    if (error) throw error;
    fs.writeFile("/tmp/print.pdf", data, function(error) {
      if (error)
        throw error;
      console.log("Write PDF successfully.");
    })
  })
});
```

`webContents.addWorkSpace(path)`

- `path` String

添加指定的路径给开发者工具栏的 workspace.必须在 DevTools 创建之后使用它：

```
mainWindow.webContents.on('devtools-opened', function() {
  mainWindow.webContents.addWorkSpace(__dirname);
});
```

`webContents.removeWorkSpace(path)`

- `path` String

从开发者工具栏的 workspace 删除指定的路径.

`webContents.openDevTools([options])`

- `options` Object (可选)
  - `detach` Boolean - 在一个新窗口打开开发者工具栏

打开开发者工具栏.

`webContents.closeDevTools()`

关闭开发者工具栏.

`webContents.isDevToolsOpened()`

返回布尔值，开发者工具栏是否打开.

`webContents.isDevToolsFocused()`

返回布尔值，开发者工具栏视图是否获得焦点.

`webContents.toggleDevTools()`

Toggles 开发者工具.

`webContents.inspectElement(x, y)`

- `x` Integer
- `y` Integer

在 (`x` , `y`) 开始检测元素.

`webContents.inspectServiceWorker()`

为 service worker 上下文打开开发者工具栏.

`webContents.send(channel[, arg1][, arg2][, ...])`

- `channel` String
- `arg` (可选)

通过 `channel` 发送异步消息给渲染进程，你也可发送任意的参数.参数应该在 JSON 内部序列化，并且此后没有函数或原形链被包括了.

渲染进程可以通过使用 `ipcRenderer` 监听 `channel` 来处理消息.

例子，从主进程向渲染进程发送消息：

```
// 主进程.
var window = null;
app.on('ready', function() {
  window = new BrowserWindow({width: 800, height: 600});
  window.loadURL('file://' + __dirname + '/index.html');
  window.webContents.on('did-finish-load', function() {
    window.webContents.send('ping', 'whoooooooh!');
  });
});
```

```
<!-- index.html -->
<html>
<body>
  <script>
    require('electron').ipcRenderer.on('ping', function(event, message) {
      console.log(message);  // Prints "whoooooooh!"
    });
  </script>
</body>
</html>
```

`webContents.enableDeviceEmulation(parameters)`

`parameters` Object, properties:

- `screenPosition` String - 指定需要模拟的屏幕 (默认 : `desktop`)
  - `desktop`
  - `mobile`
- `screenSize` Object - 设置模拟屏幕 size (screenPosition == mobile)
  - `width` Integer - 设置模拟屏幕 width
  - `height` Integer - 设置模拟屏幕 height
- `viewPosition` Object - 在屏幕放置 view (screenPosition == mobile) (默认: `{x: 0, y: 0}`)
  - `x` Integer - 设置偏移左上角的x轴
  - `y` Integer - 设置偏移左上角的y轴
- `deviceScaleFactor` Integer - 设置设备比例因子 (如果为0，默认为原始屏幕比例) (默认: `0`)
- `viewSize` Object - 设置模拟视图 size (空表示不覆盖)

  - `width` Integer - 设置模拟视图 width
  - `height` Integer - 设置模拟视图 height
- `fitToView` Boolean - 如果有必要的话，是否把模拟视图按比例缩放来适应可用空间 (默认: `false`)
- `offset` Object - 可用空间内的模拟视图偏移 (不在适应模式) (默认: `{x: 0, y: 0}`)
  - `x` Float - 设置相对左上角的x轴偏移值
  - `y` Float - 设置相对左上角的y轴偏移值
- `scale` Float - 可用空间内的模拟视图偏移 (不在适应视图模式) (默认: `1`)

使用给定的参数来开启设备模拟.

`webContents.disableDeviceEmulation()`

使用 `webContents.enableDeviceEmulation` 关闭设备模拟.

`webContents.sendInputEvent(event)`

- `event` Object
  - `type` String (必需) - 事件类型，可以是 `mouseDown` , `mouseUp` , `mouseEnter` , `mouseLeave` , `contextMenu` , `mouseWheel` , `mouseMove` , `keyDown` , `keyUp` , `char` .
  - `modifiers` Array - 事件的 modifiers 数组, 可以是 include `shift` , `control` , `alt` , `meta` , `isKeypad` , `isAutoRepeat` , `leftButtonDown` , `middleButtonDown` , `rightButtonDown` , `capsLock` , `numLock` , `left` , `right` .

向 page 发送一个输入 `event` .

对键盘事件来说，`event` 对象还有如下属性：

- `keyCode` String (必需) - 特点是将作为键盘事件发送. 可用的 key codes Accelerator.

对鼠标事件来说，`event` 对象还有如下属性：

- `x` Integer (**required**)
- `y` Integer (**required**)
- `button` String - button 按下, 可以是 `left` , `middle` , `right`
- `globalX` Integer
- `globalY` Integer
- `movementX` Integer
- `movementY` Integer
- `clickCount` Integer

对鼠标滚轮事件来说，`event` 对象还有如下属性：

- `deltaX` Integer
- `deltaY` Integer

- `wheelTicksX` Integer
- `wheelTicksY` Integer
- `accelerationRatioX` Integer
- `accelerationRatioY` Integer
- `hasPreciseScrollingDeltas` Boolean
- `canScroll` Boolean

`webContents.beginFrameSubscription(callback)`

- `callback` Function

开始订阅 提交 事件和捕获数据帧，当有 提交 事件时，使用 `callback(frameBuffer)` 调用 `callback`.

`frameBuffer` 是一个包含原始像素数据的 `Buffer`,像素数据是按照 32bit BGRA 格式有效存储的，但是实际情况是取决于处理器的字节顺序的(大多数的处理器是存放小端序的，如果是在大端序的处理器上，数据是 32bit ARGB 格式).

`webContents.endFrameSubscription()`

停止订阅帧提交事件.

`webContents.savePage(fullPath, saveType, callback)`

- `fullPath` String - 文件的完整路径.
- `saveType` String - 指定保存类型.
  - `HTMLOnly` - 只保存html.
  - `HTMLComplete` - 保存整个 page 内容.
  - `MHTML` - 保存完整的 html 为 MHTML.
- `callback` Function - `function(error) {}`.
  - `error` Error

如果保存界面过程初始化成功，返回 true.

```
win.loadURL('https://github.com');

win.webContents.on('did-finish-load', function() {
  win.webContents.savePage('/tmp/test.html', 'HTMLComplete', function(error) {
    if (!error)
      console.log("Save page successfully");
  });
});
```

实例属性

`WebContents` 对象也有下列属性:

`webContents.session`

返回这个 `webContents` 使用的 session 对象.

`webContents.hostWebContents`

返回这个 `webContents` 的父 `webContents` .

`webContents.devToolsWebContents`

获取这个 `WebContents` 的开发者工具栏的 `WebContents` .

注意: 用户不可保存这个对象，因为当开发者工具栏关闭的时候它的值为 `null` .

`webContents.debugger`

调试 API 为 remote debugging protocol 提供交替传送.

```
try {
  win.webContents.debugger.attach("1.1");
} catch(err) {
  console.log("Debugger attach failed : ", err);
};

win.webContents.debugger.on('detach', function(event, reason) {
  console.log("Debugger detached due to : ", reason);
});

win.webContents.debugger.on('message', function(event, method, params) {
  if (method == "Network.requestWillBeSent") {
    if (params.request.url == "https://www.github.com")
      win.webContents.debugger.detach();
  }
})

win.webContents.debugger.sendCommand("Network.enable");
```

`webContents.debugger.attach([protocolVersion])`

- `protocolVersion` String (可选) - 请求调试协议版本.

添加 `webContents` 调试.

`webContents.debugger.isAttached()`

返回一个布尔值，标识是否已经给 `webContents` 添加了调试.

`webContents.debugger.detach()`

删除 `webContents` 调试.

`webContents.debugger.sendCommand(method[, commandParams, callback])`

- `method` String - 方法名, 应该是由远程调试协议定义的方法.
- `commandParams` Object (可选) - 请求参数为 JSON 对象.
- `callback` Function (可选) - Response
  - `error` Object - 错误消息，标识命令失败.
  - `result` Object - 回复在远程调试协议中由 'returns'属性定义的命令描述.

发送给定命令给调试目标.

Event: 'detach'

- `event` Event
- `reason` String - 拆分调试器原因.

在调试 session 结束时发出事件.这在 `webContents` 关闭时或 `webContents` 请求开发者工具栏时发生.

Event: 'message'

- `event` Event
- `method` String - 方法名.
- `params` Object - 在远程调试协议中由 'parameters' 属性定义的事件参数.

每当调试目标发出事件时发出.

Tray

用一个 `Tray` 来表示一个图标,这个图标处于正在运行的系统的通知区 ，通常被添加到一个 context menu 上.

```
const electron = require('electron');
const app = electron.app;
const Menu = electron.Menu;
const Tray = electron.Tray;

var appIcon = null;
app.on('ready', function(){
  appIcon = new Tray('/path/to/my/icon');
  var contextMenu = Menu.buildFromTemplate([
    { label: 'Item1', type: 'radio' },
    { label: 'Item2', type: 'radio' },
    { label: 'Item3', type: 'radio', checked: true },
    { label: 'Item4', type: 'radio' }
  ]);
  appIcon.setToolTip('This is my application.');
  appIcon.setContextMenu(contextMenu);
});
```

平台限制:

- 在 Linux， 如果支持应用指示器则使用它，否则使用 `GtkStatusIcon` 代替.
- 在 Linux ，配置了只有有了应用指示器的支持, 你必须安装 `libappindicator1` 来让 tray icon 执行.
- 应用指示器只有在它拥有 context menu 时才会显示.
- 当在linux 上使用了应用指示器，将忽略点击事件.
- 在 Linux，为了让单独的 `MenuItem` 起效，需要再次调用 `setContextMenu` .例如:

```
contextMenu.items[2].checked = false;
appIcon.setContextMenu(contextMenu);
```

如果想在所有平台保持完全相同的行为，不应该依赖点击事件，而是一直将一个 context menu 添加到 tray icon.

Class: Tray

`Tray` 是一个 事件发出者.

`new Tray(image)`

- `image` NativeImage

创建一个与 `image` 相关的 icon.

事件

`Tray` 模块可发出下列事件:

注意: 一些事件只能在特定的os中运行，已经标明.

Event: 'click'

- `event` Event
  - `altKey` Boolean
  - `shiftKey` Boolean
  - `ctrlKey` Boolean
  - `metaKey` Boolean
- `bounds` Object - tray icon 的 bounds.
  - `x` Integer
  - `y` Integer
  - `width` Integer
  - `height` Integer

当tray icon被点击的时候发出事件.

注意: `bounds` 只在 OS X 和 Windows 上起效.

Event: 'right-click' *OS X Windows*

- `event` Event
  - `altKey` Boolean
  - `shiftKey` Boolean
  - `ctrlKey` Boolean
  - `metaKey` Boolean
- `bounds` Object - tray icon 的 bounds.
  - `x` Integer
  - `y` Integer
  - `width` Integer
  - `height` Integer

当tray icon被鼠标右键点击的时候发出事件.

Event: 'double-click' *OS X Windows*

- `event` Event
  - `altKey` Boolean

    - `shiftKey` Boolean
    - `ctrlKey` Boolean
    - `metaKey` Boolean
- `bounds` Object - tray icon 的 bounds.
    - `x` Integer
    - `y` Integer
    - `width` Integer
    - `height` Integer

当tray icon被双击的时候发出事件.

Event: 'balloon-show' *Windows*

当tray 气泡显示的时候发出事件.

Event: 'balloon-click' *Windows*

当tray 气泡被点击的时候发出事件.

Event: 'balloon-closed' *Windows*

当tray 气泡关闭的时候发出事件，因为超时或人为关闭.

Event: 'drop' *OS X*

当tray icon上的任何可拖动项被删除的时候发出事件.

Event: 'drop-files' *OS X*

- `event`
- `files` Array - 已删除文件的路径.

当tray icon上的可拖动文件被删除的时候发出事件.

Event: 'drag-enter' *OS X*

当一个拖动操作进入tray icon的时候发出事件.

Event: 'drag-leave' *OS X*

当一个拖动操作离开tray icon的时候发出事件. Emitted when a drag operation exits the tray icon.

Event: 'drag-end' *OS X*

当一个拖动操作在tray icon上或其它地方停止拖动的时候发出事件.

方法

`Tray` 模块有以下方法:

Note: 一些方法只能在特定的os中运行，已经标明.

`Tray.destroy()`

立刻删除 tray icon.

`Tray.setImage(image)`

- `image` NativeImage

让 `image` 与 tray icon 关联起来.

`Tray.setPressedImage(image)` *OS X*

- `image` NativeImage

当在 OS X 上按压 tray icon 的时候， 让 `image` 与 tray icon 关联起来.

`Tray.setToolTip(toolTip)`

- `toolTip` String

为 tray icon 设置 hover text.

`Tray.setTitle(title)` *OS X*

- `title` String

在状态栏沿着 tray icon 设置标题.

`Tray.setHighlightMode(highlight)` *OS X*

- `highlight` Boolean

当 tray icon 被点击的时候，是否设置它的背景色变为高亮(blue).默认为 true.

`Tray.displayBalloon(options)` *Windows*

- `options` Object
  - `icon` NativeImage
  - `title` String
  - `content` String

展示一个 tray balloon.

`Tray.popUpContextMenu([menu, position])` *OS X Windows*

- `menu` Menu (optional)
- `position` Object (可选) - 上托位置.
  - `x` Integer
  - `y` Integer

从 tray icon 上托出 context menu . 当划过 `menu` 的时候， `menu` 显示，代替 tray 的 context menu .

`position` 只在 windows 上可用，默认为 (0, 0) .

`Tray.setContextMenu(menu)`

- `menu` Menu

为这个 icon 设置 context menu .

在渲染进程（网页）内可用的模块

desktopCapturer

`desktopCapturer` 模块可用来获取可用资源，这个资源可通过 `getUserMedia` 捕获得到.

```
// 在渲染进程中.
var desktopCapturer = require('electron').desktopCapturer;

desktopCapturer.getSources({types: ['window', 'screen']}, function(error, sources) {
  if (error) throw error;
  for (var i = 0; i < sources.length; ++i) {
    if (sources[i].name == "Electron") {
      navigator.webkitGetUserMedia({
        audio: false,
        video: {
          mandatory: {
            chromeMediaSource: 'desktop',
            chromeMediaSourceId: sources[i].id,
            minWidth: 1280,
            maxWidth: 1280,
            minHeight: 720,
            maxHeight: 720
          }
        }
      }, gotStream, getUserMediaError);
      return;
    }
  }
});

function gotStream(stream) {
  document.querySelector('video').src = URL.createObjectURL(stream);
}

function getUserMediaError(e) {
  console.log('getUserMediaError');
}
```

当调用 `navigator.webkitGetUserMedia` 时创建一个约束对象，如果使用 `desktopCapturer` 的资源，必须设置 `chromeMediaSource` 为 `"desktop"` ，并且 `audio` 为 `false` .

如果你想捕获整个桌面的 audio 和 video，你可以设置 `chromeMediaSource` 为 `"screen"` ，和 `audio` 为 `true` . 当使用这个方法的时候，不可以指定一个 `chromeMediaSourceId` .

方法

`desktopCapturer` 模块有如下方法:

`desktopCapturer.getSources(options, callback)`

- `options` Object
  - `types` Array - 一个 String 数组，列出了可以捕获的桌面资源类型, 可用类型为 `screen` 和 `window` .

  - `thumbnailSize` Object (可选) - 建议缩略可被缩放的 size, 默认为 `{width: 150, height: 150}` .
- `callback` Function

发起一个请求，获取所有桌面资源，当请求完成的时候使用 `callback(error, sources)` 调用 `callback` .

`sources` 是一个 `Source` 对象数组, 每个 `Source` 表示了一个捕获的屏幕或单独窗口，并且有如下属性 :

- `id` String - 在 `navigator.webkitGetUserMedia` 中使用的捕获窗口或屏幕的 id . 格式为 `window:XX` 祸 `screen:XX` ， `XX` 是一个随机数.
- `name` String - 捕获窗口或屏幕的描述名 . 如果资源为屏幕，名字为 `Entire Screen` 或 `Screen <index>` ; 如果资源为窗口, 名字为窗口的标题.
- `thumbnail` NativeImage - 缩略图.

注意: 不能保证 `source.thumbnail` 的 size 和 `options` 中的 `thumnbailSize` 一直一致. 它也取决于屏幕或窗口的缩放比例.

ipcRenderer

`ipcRenderer` 模块是一个 EventEmitter 类的实例. 它提供了有限的方法，你可以从渲染进程向主进程发送同步或异步消息. 也可以收到主进程的相应.

查看 ipcMain 代码例子.

消息监听

`ipcRenderer` 模块有下列方法来监听事件:

`ipcRenderer.on(channel, listener)`

- `channel` String
- `listener` Function

监听 `channel` , 当有新消息到达，使用 `listener(event, args...)` 调用 `listener` .

`ipcRenderer.once(channel, listener)`

- `channel` String
- `listener` Function

为这个事件添加一个一次性 `listener` 函数.这个 `listener` 将在下一次有新消息被发送到 `channel` 的时候被请求调用，之后就被删除了.

`ipcRenderer.removeListener(channel, listener)`

- `channel` String
- `listener` Function

从指定的 `channel` 中的监听者数组删除指定的 `listener` .

`ipcRenderer.removeAllListeners([channel])`

- `channel` String (optional)

删除所有的监听者，或者删除指定 `channel` 中的全部.

发送消息

`ipcRenderer` 模块有如下方法来发送消息:

`ipcRenderer.send(channel[, arg1][, arg2][, ...])`

- `channel` String
- `arg` (可选)

通过 `channel` 向主进程发送异步消息，也可以发送任意参数.参数会被JSON序列化，之后就不会包含函数或原型链.

主进程通过使用 `ipcMain` 模块来监听 `channel`，从而处理消息.

`ipcRenderer.sendSync(channel[, arg1][, arg2][, ...])`

- `channel` String
- `arg` (可选)

通过 `channel` 向主进程发送同步消息，也可以发送任意参数.参数会被JSON序列化，之后就不会包含函数或原型链.

主进程通过使用 `ipcMain` 模块来监听 `channel`，从而处理消息, 通过 `event.returnValue` 来响应.

注意: 发送同步消息将会阻塞整个渲染进程,除非你知道你在做什么，否则就永远不要用它 .

`ipcRenderer.sendToHost(channel[, arg1][, arg2][, ...])`

- `channel` String
- `arg` (可选)

类似 `ipcRenderer.send` ，但是它的事件将发往 host page 的 `<webview>` 元素，而不是主进程.

remote

`remote` 模块提供了一种在渲染进程（网页）和主进程之间进行进程间通讯（IPC）的简便途径。

Electron中, 与GUI相关的模块（如 `dialog` , `menu` 等)只存在于主进程，而不在渲染进程中。为了能从渲染进程中使用它们，需要用 `ipc` 模块来给主进程发送进程间消息。使用 `remote` 模块，可以调用主进程对象的方法，而无需显式地发送进程间消息，这类似于 Java 的 RMI。 下面是从渲染进程创建一个浏览器窗口的例子：

```
const remote = require('electron').remote;
const BrowserWindow = remote.BrowserWindow;

var win = new BrowserWindow({ width: 800, height: 600 });
win.loadURL('https://github.com');
```

注意: 反向操作（从主进程访问渲染进程），可以使用webContents.executeJavascript.

远程对象

`remote` 模块返回的每个对象（包括函数）都代表了主进程中的一个对象（我们称之为远程对象或者远程函数）。 当调用远程对象的方法、执行远程函数或者使用远程构造器（函数）创建新对象时，其实就是在发送同步的进程间消息。

在上面的例子中， `BrowserWindow` 和 `win` 都是远程对象，然而 `new BrowserWindow` 并没有在渲染进程中创建 `BrowserWindow` 对象。 而是在主进程中创建了 `BrowserWindow` 对象，并在渲染进程中返回了对应的远程对象，即 `win` 对象。

请注意只有 可枚举属性 才能通过 remote 进行访问.

远程对象的生命周期

Electron 确保在渲染进程中的远程对象存在（换句话说，没有被垃圾收集），那主进程中的对应对象也不会被释放。 当远程对象被垃圾收集之后，主进程中的对应对象才会被取消关联。

如果远程对象在渲染进程泄露了（即，存在某个表中但永远不会释放），那么主进程中的对应对象也一样会泄露， 所以你必须小心不要泄露了远程对象。If the remote object is leaked in the renderer process (e.g. stored in a map but never freed), the corresponding object in the main process will also be leaked, so you should be very careful not to leak remote objects.

不过，主要的值类型如字符串和数字，是传递的副本。

给主进程传递回调函数

在主进程中的代码可以从渲染进程—— `remote` 模块——中接受回调函数，但是使用这个功能的时候必须非常非常小心。Code in the main process can accept callbacks from the renderer - for instance the `remote` module - but you should be extremely careful when using this feature.

首先，为了避免死锁，传递给主进程的回调函数会进行异步调用。所以不能期望主进程来获得传递过去的回调函数的返回值。First, in order to avoid deadlocks, the callbacks passed to the main process are called asynchronously. You should not expect the main process to get the return value of the passed callbacks.

比如，你不能主进程中给 `Array.map` 传递来自渲染进程的函数。

```
// 主进程 mapNumbers.js
exports.withRendererCallback = function(mapper) {
  return [1,2,3].map(mapper);
}

exports.withLocalCallback = function() {
  return exports.mapNumbers(function(x) {
    return x + 1;
  });
}
```

```
// 渲染进程
var mapNumbers = require("remote").require("./mapNumbers");

var withRendererCb = mapNumbers.withRendererCallback(function(x) {
  return x + 1;
})

var withLocalCb = mapNumbers.withLocalCallback()

console.log(withRendererCb, withLocalCb) // [true, true, true], [2, 3, 4]
```

如你所见，渲染器回调函数的同步返回值没有按预期产生，与主进程中的一模一样的回调函数的返回值不同。

其次，传递给主进程的函数会持续到主进程对他们进行垃圾回收。

例如，下面的代码第一眼看上去毫无问题。给远程对象的 `close` 事件绑定了一个回调函数：

```
remote.getCurrentWindow().on('close', function() {
  // blabla...
});
```

但记住主进程会一直保持对这个回调函数的引用，除非明确的卸载它。如果不卸载，每次重新载入窗口都会再次绑定，这样每次重启就会泄露一个回调函数。

更严重的是，由于前面安装了回调函数的上下文已经被释放，所以当主进程的 `close` 事件触发的时候，会抛出异常。

为了避免这个问题，要确保对传递给主进程的渲染器的回调函数进行清理。可以清理事件处理器，或者明确告诉主进行取消来自已经退出的渲染器进程中的回调函数。

访问主进程中的内置模块

在主进程中的内置模块已经被添加为 `remote` 模块中的属性，所以可以直接像使用 `electron` 模块一样直接使用它们。

```
const app = remote.app;
```

方法

`remote` 模块有以下方法：

`remote.require(module)`

- `module` String

返回在主进程中执行 `require(module)` 所返回的对象。

`remote.getCurrentWindow()`

返回该网页所属的 `BrowserWindow` 对象。

`remote.getCurrentWebContents()`

返回该网页的 `WebContents` 对象

`remote.getGlobal(name)`

- `name` String

返回在主进程中名为 `name` 的全局变量(即 `global[name]`) 。

`remote.process`

返回主进程中的 `process` 对象。等同于 `remote.getGlobal('process')` 但是有缓存。

webFrame

`web-frame` 模块允许你自定义如何渲染当前网页 .

例子，放大当前页到 200%.

```
var webFrame = require('electron').webFrame;

webFrame.setZoomFactor(2);
```

方法

`web-frame` 模块有如下方法:

`webFrame.setZoomFactor(factor)`

- `factor` Number - 缩放参数.

将缩放参数修改为指定的参数值.缩放参数是百分制的，所以 300% = 3.0.

`webFrame.getZoomFactor()`

返回当前缩放参数值.

`webFrame.setZoomLevel(level)`

- `level` Number - 缩放水平

将缩放水平修改为指定的水平值. 原始 size 为 0 ，并且每次增长都表示放大 20% 或缩小 20%，默认限制为原始 size 的 300% 到 50% 之间 .

`webFrame.getZoomLevel()`

返回当前缩放水平值.

`webFrame.setZoomLevelLimits(minimumLevel, maximumLevel)`

- `minimumLevel` Number
- `maximumLevel` Number

设置缩放水平的最大值和最小值.

`webFrame.setSpellCheckProvider(language, autoCorrectWord, pr`

- `language` String
- `autoCorrectWord` Boolean
- `provider` Object

为输入框或文本域设置一个拼写检查 provider .

`provider` 必须是一个对象，它有一个 `spellCheck` 方法，这个方法返回扫过的单词是否拼写正确 .

例子，使用 node-spellchecker 作为一个 provider:

```
webFrame.setSpellCheckProvider("en-US", true, {
  spellCheck: function(text) {
    return !(require('spellchecker').isMisspelled(text));
  }
});
```

`webFrame.registerURLSchemeAsSecure(scheme)`

- `scheme` String

注册 `scheme` 为一个安全的 scheme.

安全的 schemes 不会引发混合内容 warnings.例如, `https` 和 `data` 是安全的 schemes ，因为它们不能被活跃网络攻击而失效.

`webFrame.registerURLSchemeAsBypassingCSP(scheme)`

- `scheme` String

忽略当前网页内容的安全策略，直接从 `scheme` 加载.

`webFrame.registerURLSchemeAsPrivileged(scheme)`

- `scheme` String

通过资源的内容安全策略，注册 `scheme` 为安全的 scheme，允许注册 ServiceWorker并且支持 fetch API.

`webFrame.insertText(text)`

- `text` String

向获得焦点的原色插入内容 .

`webFrame.executeJavaScript(code[, userGesture])`

- `code` String
- `userGesture` Boolean (可选) - 默认为 `false`.

评估页面代码.

在浏览器窗口中，一些 HTML APIs ，例如 `requestFullScreen` ，只可以通过用户手势来使用. 设置 `userGesture` 为 `true` 可以突破这个限制.

在两种进程中都可用的模块

clipboard

`clipboard` 模块提供方法来供复制和粘贴操作 . 下面例子展示了如何将一个字符串写道 clipboard 上:

```
const clipboard = require('electron').clipboard;
clipboard.writeText('Example String');
```

在 X Window 系统上, 有一个可选的 clipboard. 你可以为每个方法使用 `selection` 来控制它:

```
clipboard.writeText('Example String', 'selection');
console.log(clipboard.readText('selection'));
```

方法

`clipboard` 模块有以下方法:

注意: 测试 APIs 已经标明，并且在将来会被删除 .

`clipboard.readText([type])`

- `type` String (可选)

以纯文本形式从 clipboard 返回内容 .

`clipboard.writeText(text[, type])`

- `text` String
- `type` String (可选)

以纯文本形式向 clipboard 添加内容 .

`clipboard.readHtml([type])`

- `type` String (可选)

返回 clipboard 中的标记内容.

`clipboard.writeHtml(markup[, type])`

- `markup` String
- `type` String (可选)

向 clipboard 添加 `markup` 内容 .

`clipboard.readImage([type])`

- `type` String (可选)

从 clipboard 中返回 NativeImage 内容.

`clipboard.writeImage(image[, type])`

- `image` NativeImage
- `type` String (可选)

向 clipboard 中写入 `image` .

`clipboard.readRtf([type])`

- `type` String (可选)

从 clipboard 中返回 RTF 内容.

`clipboard.writeRtf(text[, type])`

- `text` String
- `type` String (可选)

向 clipboard 中写入 RTF 格式的 `text` .

`clipboard.clear([type])`

- `type` String (可选)

清空 clipboard 内容.

`clipboard.availableFormats([type])`

- `type` String (可选)

返回 clipboard 支持的格式数组 .

`clipboard.has(data[, type])` *Experimental*

- `data` String
- `type` String (可选)

返回 clipboard 是否支持指定 `data` 的格式.

```
console.log(clipboard.has('<p>selection</p>'));
```

`clipboard.read(data[, type])` *Experimental*

- `data` String
- `type` String (可选)

读取 clipboard 的 `data` .

`clipboard.write(data[, type])`

- `data` Object
  - `text` String
  - `html` String
  - `image` NativeImage
- `type` String (可选)

```
clipboard.write({text: 'test', html: "<b>test</b>"});
```

向 clipboard 写入 `data` .

crashReporter

`crash-reporter` 模块开启发送应用崩溃报告.

下面是一个自动提交崩溃报告给服务器的例子：

```
const crashReporter = require('electron').crashReporter;

crashReporter.start({
  productName: 'YourName',
  companyName: 'YourCompany',
  submitURL: 'https://your-domain.com/url-to-submit',
  autoSubmit: true
});
```

可以使用下面的项目来创建一个服务器，用来接收和处理崩溃报告：

- socorro
- mini-breakpad-server

方法

`crash-reporter` 模块有如下方法:

`crashReporter.start(options)`

- `options` Object
  - `companyName` String
  - `submitURL` String - 崩溃报告发送的路径，以post方式.
  - `productName` String (可选) - 默认为 `Electron` .
  - `autoSubmit` Boolean - 是否自动提交. 默认为 `true` .
  - `ignoreSystemCrashHandler` Boolean - 默认为 `false` .
  - `extra` Object - 一个你可以定义的对象，附带在崩溃报告上一起发送 . 只有字符串属性可以被正确发送，不支持嵌套对象.

只可以在使用其它 `crashReporter` APIs 之前使用这个方法.

注意: 在 OS X, Electron 使用一个新的 `crashpad` 客户端, 与 Windows 和 Linux 的 `breakpad` 不同. 为了开启崩溃点搜集，你需要在主进程和其它每个你需要搜集崩溃报告的渲染进程中调用 `crashReporter.start` API 来初始化 `crashpad` .

`crashReporter.getLastCrashReport()`

返回最后一个崩溃报告的日期和 ID.如果没有过崩溃报告发送过来，或者还没有开始崩溃报告搜集，将返回 `null` .

`crashReporter.getUploadedReports()`

返回所有上载的崩溃报告，每个报告包含了上载日期和 ID.

crash-reporter Payload

崩溃报告将发送下面 `multipart/form-data` `POST` 型的数据给 `submitURL` :

- `ver` String - Electron 版本.
- `platform` String - 例如 'win32'.
- `process_type` String - 例如 'renderer'.
- `guid` String - 例如 '5e1286fc-da97-479e-918b-6bfb0c3d1c72'
- `_version` String - `package.json` 版本.
- `_productName` String - `crashReporter` `options` 对象中的产品名字.
- `prod` String - 基础产品名字. 这种情况为 Electron.
- `_companyName` String - `crashReporter` `options` 对象中的公司名字.
- `upload_file_minidump` File - 崩溃报告按照 `minidump` 的格式.
- `crashReporter` 中的 `extra` 对象的所有等级和一个属性. `options` object

nativeImage

在 Electron 中, 对所有创建 images 的 api 来说, 你可以使用文件路径或 `nativeImage` 实例. 如果使用 `null` ，将创建一个空的image 对象.

例如, 当创建一个 tray 或设置窗口的图标时候，你可以使用一个字符串的图片路径 :

```
var appIcon = new Tray('/Users/somebody/images/icon.png');
var window = new BrowserWindow({icon: '/Users/somebody/images/window.png'});
```

或者从剪切板中读取图片，它返回的是 `nativeImage` :

```
var image = clipboard.readImage();
var appIcon = new Tray(image);
```

支持的格式

当前支持 `PNG` 和 `JPEG` 图片格式. 推荐 `PNG` ，因为它支持透明和无损压缩.

在 Windows, 你也可以使用 `ICO` 图标的格式.

高分辨率图片

如果平台支持 high-DPI，你可以在图片基础路径后面添加 `@2x` ，可以标识它为高分辨率的图片.

例如，如果 `icon.png` 是一个普通图片并且拥有标准分辨率，然后 `icon@2x.png` 将被当作高分辨率的图片处理，它将拥有双倍 DPI 密度.

如果想同时支持展示不同分辨率的图片，你可以将拥有不同size 的图片放在同一个文件夹下，不用 DPI 后缀.例如 :

```
images/
├── icon.png
├── icon@2x.png
└── icon@3x.png
```

```
var appIcon = new Tray('/Users/somebody/images/icon.png');
```

也支持下面这些 DPI 后缀:

- `@1x`

- `@1.25x`
- `@1.33x`
- `@1.4x`
- `@1.5x`
- `@1.8x`
- `@2x`
- `@2.5x`
- `@3x`
- `@4x`
- `@5x`

模板图片

模板图片由黑色和清色(和一个 alpha 通道)组成. 模板图片不是单独使用的，而是通常和其它内容混合起来创建期望的最终效果.

最常见的用力是将模板图片用到菜单栏图片上，所以它可以同时适应亮、黑不同的菜单栏.

注意: 模板图片只在 OS X 上可用.

为了将图片标识为一个模板图片，它的文件名应当以 `Template` 结尾. 例如:

- `xxxTemplate.png`
- `xxxTemplate@2x.png`

方法

`nativeImage` 类有如下方法:

`nativeImage.createEmpty()`

创建一个空的 `nativeImage` 实例.

`nativeImage.createFromPath(path)`

- `path` String

从指定 `path` 创建一个新的 `nativeImage` 实例 .

`nativeImage.createFromBuffer(buffer[, scaleFactor])`

- `buffer` Buffer
- `scaleFactor` Double (可选)

从 `buffer` 创建一个新的 `nativeImage` 实例 .默认 `scaleFactor` 是 1.0.

`nativeImage.createFromDataURL(dataURL)`

- `dataURL` String

从 `dataURL` 创建一个新的 `nativeImage` 实例 .

实例方法

`nativeImage` 有如下方法:

```
const nativeImage = require('electron').nativeImage;

var image = nativeImage.createFromPath('/Users/somebody/images/icon.png');
```

`image.toPng()`

返回一个 Buffer ，它包含了图片的 `PNG` 编码数据.

`image.toJpeg(quality)`

- `quality` Integer (必须) - 在 0 - 100 之间.

返回一个 Buffer ，它包含了图片的 `JPEG` 编码数据.

`image.toDataURL()`

返回图片数据的 URL.

`image.getNativeHandle()` *OS X*

返回一个保存了 c 指针的 Buffer 来潜在处理原始图像.在OS X, 将会返回一个 `NSImage` 指针实例.

注意那返回的指针是潜在原始图像的弱指针，而不是一个复制，你必须 确保与 `nativeImage` 的关联不间断 .

`image.isEmpty()`

返回一个 boolean ，标识图片是否为空.

`image.getSize()`

返回图片的 size.

`image.setTemplateImage(option)`

- `option` Boolean

将图片标识为模板图片.

`image.isTemplateImage()`

返回一个 boolean ，标识图片是否是模板图片.

screen

`screen` 模块检索屏幕的 size，显示，鼠标位置等的信息.在 `app` 模块的 `ready` 事件触发之前不可使用这个模块.

`screen` 是一个 EventEmitter.

注意: 在渲染进程 / 开发者工具栏, `window.screen` 是一个预设值的 DOM 属性, 所以这样写 `var screen = require('electron').screen` 将不会工作. 在我们下面的例子, 我们取代使用可变名字的 `electronScreen` . 一个例子，创建一个充满真个屏幕的窗口 :

```
const electron = require('electron');
const app = electron.app;
const BrowserWindow = electron.BrowserWindow;

var mainWindow;

app.on('ready', function() {
  var electronScreen = electron.screen;
  var size = electronScreen.getPrimaryDisplay().workAreaSize;
  mainWindow = new BrowserWindow({ width: size.width, height: size.height });
});
```

另一个例子，在次页外创建一个窗口:

```
const electron = require('electron');
const app = electron.app;
const BrowserWindow = electron.BrowserWindow;

var mainWindow;

app.on('ready', function() {
  var electronScreen = electron.screen;
  var displays = electronScreen.getAllDisplays();
  var externalDisplay = null;
  for (var i in displays) {
    if (displays[i].bounds.x != 0 || displays[i].bounds.y != 0) {
      externalDisplay = displays[i];
      break;
    }
  }

  if (externalDisplay) {
    mainWindow = new BrowserWindow({
      x: externalDisplay.bounds.x + 50,
      y: externalDisplay.bounds.y + 50
    });
  }
});
```

`Display` 对象

`Display` 对象表示了物力方式连接系统. 一个伪造的 `Display` 或许存在于一个无头系统中，或者一个 `Display` 相当于一个远程的、虚拟的 display.

- `display` object
    - `id` Integer - 与display 相关的唯一性标志.
    - `rotation` Integer - 可以是 0, 1, 2, 3, 每个代表了屏幕旋转的度数 0, 90, 180, 270.
    - `scaleFactor` Number - Output device's pixel scale factor.
    - `touchSupport` String - 可以是 `available` , `unavailable` , `unknown` .
    - `bounds` Object
    - `size` Object
    - `workArea` Object
    - `workAreaSize` Object

事件

`screen` 模块有如下事件:

Event: 'display-added'

返回:

- `event` Event
- `newDisplay` Object

当添加了 `newDisplay` 时发出事件

Event: 'display-removed'

返回:

- `event` Event
- `oldDisplay` Object

当移出了 `oldDisplay` 时发出事件

Event: 'display-metrics-changed'

返回:

- `event` Event
- `display` Object
- `changedMetrics` Array

当一个 `display` 中的一个或更多的 metrics 改变时发出事件. `changedMetrics` 是一个用来描述这个改变的数组.可能的变化为 `bounds` , `workArea` , `scaleFactor` 和 `rotation` .

方法

`screen` 模块有如下方法:

`screen.getCursorScreenPoint()`

返回当前鼠标的绝对路径 .

`screen.getPrimaryDisplay()`

返回最主要的 display.

`screen.getAllDisplays()`

返回一个当前可用的 display 数组.

`screen.getDisplayNearestPoint(point)`

- `point` Object
  - `x` Integer
  - `y` Integer

返回离指定点最近的 display.

`screen.getDisplayMatching(rect)`

- `rect` Object
  - `x` Integer
  - `y` Integer
  - `width` Integer
  - `height` Integer

返回与提供的边界范围最密切相关的 display.

shell

`shell` 模块提供了集成其他桌面客户端的关联功能.

在用户默认浏览器中打开URL的示例:

```
var shell = require('shell');

shell.openExternal('https://github.com');
```

Methods

`shell` 模块包含以下函数:

`shell.showItemInFolder(fullPath)`

- `fullPath` String

打开文件所在文件夹,一般情况下还会选中它.

`shell.openItem(fullPath)`

- `fullPath` String

以默认打开方式打开文件.

`shell.openExternal(url)`

- `url` String

以系统默认设置打开外部协议.(例如,mailto: somebody@somewhere.io会打开用户默认的邮件客户端)

`shell.moveItemToTrash(fullPath)`

- `fullPath` String

删除指定路径文件,并返回此操作的状态值(boolean类型).

`shell.beep()`

播放 beep 声音.

开发

编码规范

以下是 Electron 项目的编码规范。

C++ 和 Python

对于 C++ 和 Python，我们遵循 Chromium 的编码规范。你可以使用 `script/cpplint.py` 来检验文件是否符合要求。

我们目前使用的 Pyhton 版本是 Python 2.7。

C++ 代码中用到了许多 Chromium 中的接口和数据类型，所以希望你能熟悉它们。Chromium 中的重要接口和数据结构就是一篇不错的入门文档，里面提到了一些特殊类型、域内类型（退出作用域时自动释放内存）、日志机制，等等。

CoffeeScript

对于 CoffeeScript，我们遵循 GitHub 的编码规范 及以下规则:

- 文件不要以换行符结尾，我们要遵循 Google 的编码规范。
- 文件名使用 `-` 而不是 `_` 来连接单词，比如 `file-name.coffee` 而不是 `file_name.coffee`，这是沿用 github/atom 模块的命名方式（ `module-name` ）。这条规则仅适用于 `.coffee` 文件。

API 命名

当新建一个 API 时，我们倾向于使用 getters 和 setters 而不是 jQuery 单函数的命名方式，比如 `.getText()` 和 `.setText(text)` 而不是 `.text([text])`。这里有关于该规则的讨论记录。

源码目录结构

Electron 的源代码主要依据 Chromium 的拆分约定被拆成了许多部分。

为了更好地理解源代码，您可能需要了解一下 Chromium 的多进程架构。

源代码的结构

```
Electron
├──atom - Electron 的源代码
|  ├── app - 系统入口代码
|  ├── browser - 包含了主窗口、UI 和其他所有与主进程有关的东西，它会告诉渲染进程如何管理页面
|  |   ├── lib - 主进程初始化代码中 JavaScript 部分的代码
|  |   ├── ui - 不同平台上 UI 部分的实现
|  |   |   ├── cocoa - Cocoa 部分的源代码
|  |   |   ├── gtk - GTK+ 部分的源代码
|  |   |   └── win - Windows GUI 部分的源代码
|  |   ├── default_app - 在没有指定 app 的情况下 Electron 启动时默认显示的页面
|  |   ├── api - 主进程 API 的实现
|  |   |   └── lib - API 实现中 Javascript 部分的代码
|  |   ├── net - 网络相关的代码
|  |   ├── mac - 与 Mac 有关的 Objective-C 代码
|  |   └── resources - 图标，平台相关的文件等
|  ├── renderer - 运行在渲染进程中的代码
|  |   ├── lib - 渲染进程初始化代码中 JavaScript 部分的代码
|  |   └── api - 渲染进程 API 的实现
|  |       └── lib - API 实现中 Javascript 部分的代码
|  └── common - 同时被主进程和渲染进程用到的代码，包括了一些用来将 node 的事件循环
|      |        整合到 Chromium 的事件循环中时用到的工具函数和代码
|      ├── lib - 同时被主进程和渲染进程使用到的 Javascript 初始化代码
|      └── api - 同时被主进程和渲染进程使用到的 API 的实现以及 Electron 内置模块的基础设施
|          └── lib - API 实现中 Javascript 部分的代码
├── chromium_src - 从 Chromium 项目中拷贝来的代码
├── docs - 英语版本的文档
├── docs-translations - 各种语言版本的文档翻译
├── spec - 自动化测试
├── atom.gyp - Electron 的构建规则
└── common.gypi - 为诸如 `node` 和 `breakpad` 等其他组件准备的编译设置和构建规则
```

其他目录的结构

- **script** - 用于诸如构建、打包、测试等开发用途的脚本
- **tools** - 在 gyp 文件中用到的工具脚本，但与 `script` 目录不同， 该目录中的脚本不应该被用户直接调用
- **vendor** - 第三方依赖项的源代码，为了防止人们将它与 Chromium 源码中的同名目录相混淆， 在这里我们不使用 `third_party` 作为目录名
- **node_modules** - 在构建中用到的第三方 node 模块
- **out** - `ninja` 的临时输出目录
- **dist** - 由脚本 `script/create-dist.py` 创建的临时发布目录
- **external_binaries** - 下载的不支持通过 `gyp` 构建的预编译第三方框架

Electron 和 NW.js (原名 node-webkit) 在技术上的差异

备注：**Electron** 的原名是 **Atom Shell**。

与 NW.js 相似，Electron 提供了一个能通过 JavaScript 和 HTML 创建桌面应用的平台，同时集成 Node 来授予网页访问底层系统的权限。

但是这两个项目也有本质上的区别，使得 Electron 和 NW.js 成为两个相互独立的产品。

1. 应用的入口

在 NW.js 中，一个应用的主入口是一个页面。你在 `package.json` 中指定一个主页面，它会作为应用的主窗口被打开。

在 Electron 中，入口是一个 JavaScript 脚本。不同于直接提供一个URL，你需要手动创建一个浏览器窗口，然后通过 API 加载 HTML 文件。你还可以监听窗口事件，决定何时让应用退出。

Electron 的工作方式更像 Node.js 运行时。 Electron 的 APIs 更加底层，因此你可以用它替代 PhantomJS 做浏览器测试。

2. 构建系统

为了避免构建整个 Chromium 带来的复杂度，Electron 通过 `libchromiumcontent` 来访问 Chromium 的 Content API。`libchromiumcontent` 是一个独立的、引入了 Chromium Content 模块及其所有依赖的共享库。用户不需要一个强劲的机器来构建 Electron。

3. Node 集成

在 NW.js，网页中的 Node 集成需要通过给 Chromium 打补丁来实现。但在 Electron 中，我们选择了另一种方式：通过各个平台的消息循环与 libuv 的循环集成，避免了直接在 Chromium 上做改动。你可以看 `node_bindings` 来了解这是如何完成的。

4. 多上下文

如果你是有经验的 NW.js 用户，你应该会熟悉 Node 上下文和 web 上下文的概念。这些概念的产生源于 NW.js 的实现方式。

通过使用 Node 的多上下文特性，Electron不需要在网页中引入新的 JavaScript 上下文。

Build System Overview

Electron 使用 gyp 来生成项目 ，使用 ninja 来构建项目. 项目配置可以在 `.gyp` 和 `.gypi` 文件中找到.

Gyp 文件

下面的 `gyp` 文件包含了构建 Electron 的主要规则 :

- `atom.gyp` 定义了 Electron 它自己是怎样被构建的.
- `common.gypi` 调整 node 的构建配置，来让它结合 Chromium 一起构建.
- `vendor/brightray/brightray.gyp` 定义了 `brightray` 是如何被构建的，并且包含了默认配置来连接到 Chromium.
- `vendor/brightray/brightray.gypi` 包含了常用的创建配置.

创建组件

在 Chromium 还是一个相当大的项目的时候，最后链接阶段会花了好几分钟，这让开发变得很困难. 为了解决这个困难，Chromium 引入了 "component build" ，这让每个创建的组建都是分隔开的共享库，让链接更快，但是这浪费了文件大小和性能.

在 Electron 中，我们采用了一个非常相似的方法 : 在创建 `Debug` , 二进制文件会被链接进入一个 Chromium 组件的共享版本库来达到快速链接; 在创建 `Release` , 二进制文件会被链接进入一个静态版本库, 所以我们可以有最小的二进制文件size和最佳的体验.

Minimal Bootstrapping

在运行 bootstrap 脚本的时候，所有的 Chromium 预编译二进制文件会被下载.默认静态库和共享库会被下载，并且项目的最后大小会在 800MB 到 2GB 之间，这取决于平台类型.

默认， `libchromiumcontent` 是从 Amazon Web Services 上下载下来的.如果设置了 `LIBCHROMIUMCONTENT_MIRROR` 环境变量，bootstrap脚本会从这里下载下来. `libchromiumcontent-qiniu-mirror` 是 `libchromiumcontent` 的映射.如果你不能连接 AWS，你可以切换下载路径： `export LIBCHROMIUMCONTENT_MIRROR=http://7xk3d2.dl1.z0.glb.clouddn.com/` 如果只是想快速搭建一个 Electron 的测试或开发环境，可以通过 `--dev` 参数只下载共享版本库:

```
$ ./script/bootstrap.py --dev
$ ./script/build.py -c D
```

Two-Phase Project Generation

在 `Release` 和 `Debug` 构建的时候后，Electron 链接了不同配置的库 .然而 `gyp` 不支持为不同的配置文件进行不同的链接设置.

为了规避这个问题，Electron 在运行 `gyp` 的时候，使用了一个 `gyp` 的变量 `libchromiumcontent_component` 来控制应该使用哪个链接设置，并且只生成一个目标.

Target Names

与大多数的项目不同，它们使用 `Release` 和 `Debug` 作为目标名字，而 Electron 使用使用的是 `R` 和 `D`.这是因为如果只定义了一个 `Release` 或 `Debug` 构建配置，`gyp` 会随机崩溃，并且在同一时候，Electron 只生成一个目标，如上所述.

这只对开发者可用，如果想重新构建 Electron ，将不会成功.

Build Instructions (OS X)

遵循下面的引导，在 OS X 上构建 Electron .

前提

- OS X >= 10.8
- Xcode >= 5.1
- node.js (外部)

如果你通过 Homebrew 使用 Python 下载，需要安装下面的 Python 模块:

- pyobjc

获取代码

```
$ git clone https://github.com/electron/electron.git
```

Bootstrapping

bootstrap 脚本也是必要下载的构建依赖，来创建项目文件.注意我们使用的是 ninja 来构建 Electron，所以没有生成 Xcode 项目.

```
$ cd electron
$ ./script/bootstrap.py -v
```

构建

创建 `Release` 、 `Debug` target:

```
$ ./script/build.py
```

可以只创建 `Debug` target:

```
$ ./script/build.py -c D
```

创建完毕, 可以在 `out/D` 下面找到 `Electron.app` .

32位支持

在 OS X 上，构建 Electron 只支持 64位的，不支持 32位的 .

测试

测试你的修改是否符合项目代码风格，使用:

```
$ ./script/cpplint.py
```

测试有效性使用:

```
$ ./script/test.py
```

Build Instructions (Windows)

遵循下面的引导，在 Windows 上构建 Electron .

前提

- Windows 7 / Server 2008 R2 or higher
- Visual Studio 2013 with Update 4 - download VS 2013 Community Edition for free.
- Python 2.7
- Node.js
- Git

如果你现在还没有安装 Windows , modern.ie 有一个 timebombed 版本的 Windows ，你可以用它来构建 Electron.

构建 Electron 完全的依赖于命令行，并且不可通过 Visual Studio. 可以使用任何的编辑器来开发 Electron ，未来会支持 Visual Studio.

注意: 虽然 Visual Studio 不是用来构建的，但是它仍然 必须的 ，因为我们需要它提供的构建工具栏.

注意: Visual Studio 2015 不可用. 请确定使用 MSVS **2013**.

获取代码

```
$ git clone https://github.com/electron/electron.git
```

Bootstrapping

bootstrap 脚本也是必要下载的构建依赖，来创建项目文件.注意我们使用的是 `ninja` 来构建 Electron，所以没有生成 Visual Studio 项目.

```
$ cd electron
$ python script\bootstrap.py -v
```

构建

创建 `Release` 、 `Debug` target:

```
$ python script\build.py
```

可以只创建 `Debug` target:

```
$ python script\build.py -c D
```

创建完毕, 可以在 `out/D` (debug target) 或 `out\R` (release target) 下面找到 `electron.exe` .

64bit Build

为了构建64位的 target,在运行 bootstrap 脚本的时候需要使用 `--target_arch=x64` :

```
$ python script\bootstrap.py -v --target_arch=x64
```

其他构建步骤完全相同.

Tests

测试你的修改是否符合项目代码风格，使用:

```
$ python script\cpplint.py
```

测试有效性使用:

```
$ python script\test.py
```

在构建 debug 时为 Tests包含原生模块 (例如 `runas`) 将不会执行(详情 #2558), 但是它们在构建 release 会起效.

运行 release 构建使用 :

```
$ python script\test.py -R
```

解决问题

Command xxxx not found

如果你遇到了一个错误，类似 `Command xxxx not found`, 可以尝试使用 `VS2012 Command Prompt` 控制台来执行构建脚本 .

Fatal internal compiler error: C1001

确保你已经安装了 Visual Studio 的最新安装包 .

Assertion failed: ((handle))->activecnt >= 0

如果在 Cygwin 下构建的，你可能会看到 `bootstrap.py` 失败并且附带下面错误 :

```
Assertion failed: ((handle))->activecnt >= 0, file src\win\pipe.c, line 1430

Traceback (most recent call last):
  File "script/bootstrap.py", line 87, in <module>
    sys.exit(main())
  File "script/bootstrap.py", line 22, in main
    update_node_modules('.')
  File "script/bootstrap.py", line 56, in update_node_modules
    execute([NPM, 'install'])
  File "/home/zcbenz/codes/raven/script/lib/util.py", line 118, in execute
    raise e
subprocess.CalledProcessError: Command '['npm.cmd', 'install']' returned non-zero exit st
```

这是由同时使用 Cygwin Python 和 Win32 Node 造成的 bug.解决办法就是使用 Win32 Python 执行 bootstrap 脚本 (假定你已经在目录 `C:\Python27` 下安装了 Python):

```
$ /cygdrive/c/Python27/python.exe script/bootstrap.py
```

LNK1181: cannot open input file 'kernel32.lib'

重新安装 32位的 Node.js.

Error: ENOENT, stat 'C:\Users\USERNAME\AppData\Roaming\npm'

简单创建目录 应该可以解决问题:

```
$ mkdir ~\AppData\Roaming\npm
```

node-gyp is not recognized as an internal or external command

如果你使用 Git Bash 来构建，或许会遇到这个错误，可以使用 PowerShell 或 VS2012 Command Prompt 来代替 .

Build Instructions (Linux)

遵循下面的引导，在 Linux 上构建 Electron .

Prerequisites

- Python 2.7.x. 一些发行版如 CentOS 仍然使用 Python 2.6.x ，所以或许需要 check 你的 Python 版本，使用 `python -V` .
- Node.js v0.12.x. 有很多方法来安装 Node. 可以从 Node.js下载原文件并且编译它 .也可以作为一个标准的用户在 home 目录下安装 node .或者尝试使用仓库 NodeSource.
- Clang 3.4 或更新的版本.
- GTK+开发头文件和libnotify.

在 Ubuntu, 安装下面的库 :

```
$ sudo apt-get install build-essential clang libdbus-1-dev libgtk2.0-dev \
                       libnotify-dev libgnome-keyring-dev libgconf2-dev \
                       libasound2-dev libcap-dev libcups2-dev libxtst-dev \
                       libxss1 libnss3-dev gcc-multilib g++-multilib
```

在 Fedora, 安装下面的库 :

```
$ sudo yum install clang dbus-devel gtk2-devel libnotify-devel libgnome-keyring-devel \
                   xorg-x11-server-utils libcap-devel cups-devel libXtst-devel \
                   alsa-lib-devel libXrandr-devel GConf2-devel nss-devel
```

其它版本的也许提供了相似的包来安装，通过包管理器，例如 pacman. 或一个可以编译源文件的.

使用虚拟机

如果在虚拟机上构建 Electron，你需要一个固定大小的设备，至少需要 25 gigabytes .

获取代码

```
$ git clone https://github.com/electron/electron.git
```

Bootstrapping

bootstrap 脚本也是必要下载的构建依赖，来创建项目文件.需要使用 Python 2.7.x 来让脚本成功执行.正确下载文件会花费较长的时间. 注意我们使用的是 `ninja` 来构建 Electron，所以没有生成 `Makefile` 项目.

```
$ cd electron
$ ./script/bootstrap.py -v
```

交叉编译

如果想创建一个 `arm` target ，应当还要下载下面的依赖：

```
$ sudo apt-get install libc6-dev-armhf-cross linux-libc-dev-armhf-cross \
                       g++-arm-linux-gnueabihf
```

为了编译 `arm` 或 `ia32` targets, 你应当为 `bootstrap.py` 脚本使用 `--target_arch` 参数:

```
$ ./script/bootstrap.py -v --target_arch=arm
```

构建

创建 `Release` 、 `Debug` target:

```
$ ./script/build.py
```

这个脚本也许会在目录 `out/R` 下创建一个巨大的可执行的 Electron . 文件大小或许会超过 1.3 gigabytes. 原因是 Release target 二进制文件包含了 调试符号 .运行 `create-dist.py` 脚本来减小文件的 size :

```
$ ./script/create-dist.py
```

这会在 `dist` 目录下创建一个有大量小文件的工作空间. 运行 create-dist.py 脚本之后, 或许你想删除仍然在 `out/R` 下的 1.3+ gigabyte 二进制文件.

可以只创建 `Debug` target:

```
$ ./script/build.py -c D
```

创建完毕, 可以在 `out/D` 下面找到 `electron` .

Cleaning

删除构建文件：

```
$ ./script/clean.py
```

解决问题

确保你已经安装了所有的依赖 .

Error While Loading Shared Libraries: libtinfo.so.5

预构建的 `clang` 会尝试链接到 `libtinfo.so.5`. 取决于 host 架构, 适当的使用 `libncurses`:

```
$ sudo ln -s /usr/lib/libncurses.so.5 /usr/lib/libtinfo.so.5
```

Tests

测试你的修改是否符合项目代码风格，使用:

```
$ ./script/cpplint.py
```

测试有效性使用:

```
$ ./script/test.py
```

Setting Up Symbol Server in Debugger

调试 symbols 让你有更好的调试 sessions. 它们有可执行的动态库的函数信息，并且提供信息来获得洁净的呼叫栈. 一个 Symbol 服务器允许调试器自动加载正确的 symbols, 二进制文件 和 资源文件，不用再去强制用户下载巨大的调试文件. 服务器函数类似 Microsoft's symbol server ，所以这里的记录可用.

注意，因为公众版本的 Electron 构建是最优化的，调试不一定一直简单.调试器将不会给显示出所有变量内容，并且因为内联，尾调用，和其它编译器优化，执行路径会看起来很怪异 . 唯一的解决办法是搭建一个不优化的本地构建.

Electron 使用的官方 symbol 服务器地址为 `http://54.249.141.255:8086/atom-shell/symbols` . 你不能直接访问这个路径，必须将其添加到你的调试工具的 symbol 路径上.在下面的例子中，使用了一个本地缓存目录来避免重复从服务器获取 PDB. 在你的电脑上使用一个恰当的缓存目录来代替 `c:\code\symbols` .

Using the Symbol Server in Windbg

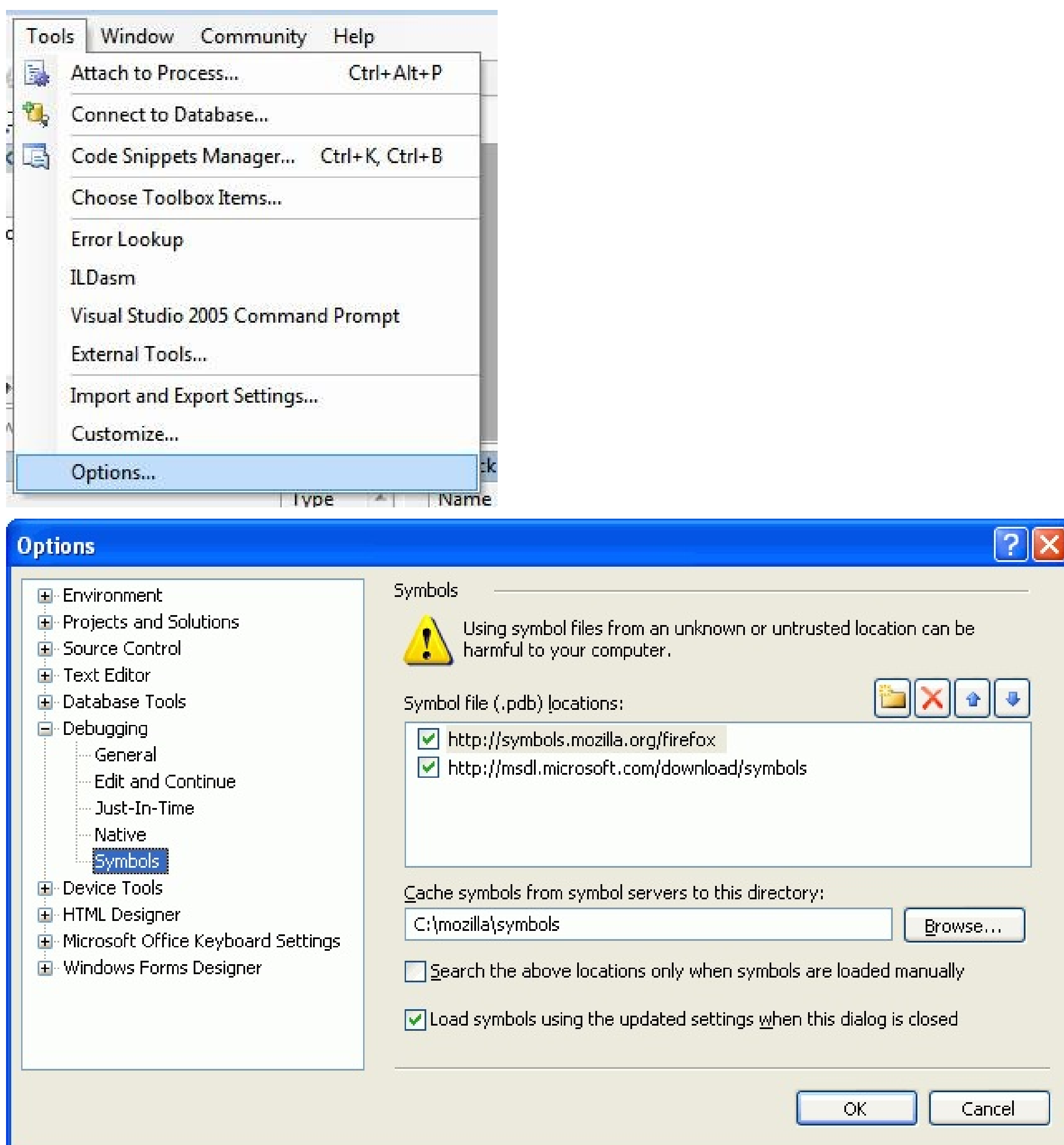
Windbg symbol 路径被配制为一个限制带星号字符的字符串. 要只使用 Electron 的 symbol 服务器, 将下列记录添加到你的 symbol 路径 (注意: 如果你愿意使用一个不同的地点来下载 symbols，你可以在你的电脑中使用任何可写的目录来代替 `c:\code\symbols`):

```
SRV*c:\code\symbols\*http://54.249.141.255:8086/atom-shell/symbols
```

使用 Windbg 菜单或通过输入 `.sympath` 命令，在环境中设置一个 `_NT_SYMBOL_PATH` 字符串. 如果你也想从微软的 symbol 服务器获得 symbols ，你应当首先将它们先列出来：

```
SRV*c:\code\symbols\*http://msdl.microsoft.com/download/symbols;SRV*c:\code\symbols\*http
```

在 Visual Studio 中使用 symbol 服务器



Troubleshooting: Symbols will not load

在 Windbg 中输入下列命令，打印出未什么 symbols 没有加载：

```
> !sym noisy
> .reload /f chromiumcontent.dll
```